Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Ingeniería en Informática

Sistemas Operativos

Trabajo Práctico N° 3

ALGORITMO DE ADMINISTRACIÓN DE MEMORIA FIRST FIT

	Nicanor Casas
Equipo	Graciela De Luca
	Waldo Valiente
Docente	Gerardo Puyo
	Martín Cortina

GRUPO: 6

	Alumnos	
Apellido	Nombre	DNI
Mosso	Juan Pablo	33.040.517
Martin	Sergio Miguel	32.691.890
Vicente	Fernando	30.047.687

Acreditación:

Instancia	Fecha	Calificación
Entrega Final	25/11/2009	

-- 2009 --

Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Ingeniería en Informática

Sistemas Operativos

Trabajo Práctico N° 3

ALGORITMO DE ADMINISTRACIÓN DE MEMORIA FIRST FIT

	Nicanor Casas
Equipo	Graciela De Luca
	Waldo Valiente
Docente	Gerardo Puyo
	Martín Cortina

GRUPO: 6

	Alumnos	
Apellido	Nombre	DNI
Mosso	Juan Pablo	33.040.517
Martin	Sergio Miguel	32.691.890
Vicente	Fernando	30.047.687

Acreditación:

Instancia	Fecha	Calificación
Entrega Final	25/11/2009	

-- 2009 --

Observaciones:

ÍNDICE

ÍNDICE 5
INTRODUCCIÓN 6
  OBJETIVOS Y ALCANCE DEL TRABAJO PRÁCTICO 7
  SOBRE EL FORMATO DE ENTREGA 7
MEJORAS PREVIAS 8
ALGORITMO DE ADMINISTRACIÓN DE MEMORIA 16
IMPLEMENTACIÓN 32
FORMATO ELF 55
PARTE FINAL 65
  CONCLUSIÓN 66

INTRODUCCIÓN

Objetivos y Alcance del Trabajo Práctico

El objetivo general del trabajo práctico es desarrollar algoritmos de partición fija y variable para administración de memoria. . Como objetivos implícitos se encuentran:

- Desarrollar los algoritmos necesarios para que funcione partición fija y variable de memoria para procesos nivel 3 del SODIUM.
- No utilizar memoria swap.
- La partición variable puede permitir al proceso agrandar la misma si no le queda mas espacio hasta tanto se llegue al límite

Sobre el formato de entrega

Esta entrega de documentación sobre el Trabajo Practico realizado está dividido en cuatro partes referidas cada una de ellas a una etapa diferente del desarrollo del trabajo práctico.

Esta división facilita por una parte mantener la cronología de los cambios y por otro lado hace mucho mas simple enumerar las modificaciones hechas en cada uno de ellos, permitiendo una más fácil redacción y también una fácil lectura por parte de los profesores de la cátedra y eventualmente el alumnado.

La primera parte (Mejoras Previas) contiene todos los cambios efectuados previos a la resolución misma del trabajo práctico. Algunos solicitados por la cátedra tras evaluar el TP1, otros necesarios para la resolución del TP3, otros son mejoras que decidimos hacerle al Sodium por nuestra cuenta y finalmente cambios de limpieza de código muerto.

La segunda parte (Algoritmo de Administración de Memoria) detallará la creación y funcionamiento del algoritmo de administración de particiones fijas y variables en sí creado por el grupo, fuera del contexto del sodium, como desarrollo teórico

La tercera parte (Implementación) expondrá detalladamente todos los pasos efectuados para la migración del viejo administrador de memoria hacia el nuevo y la manera en que ambos kernel y programas usuario lo pueden usar indistintamente.

La cuarta parte (Formato ELF) incluirá una explicación sobre como lo implementamos en el sodium para su utilización para la correcta ubicación del Heap y determinación del tamaño del proceso. Además se incluirá un apartado referido al swap y la utilidad del formato ELF para este aspecto.

Por último habrá una parte final explicando el estado final del Sodium y las conclusiones del grupo sobre el Trabajo Practico y sobre el sistema operativo en sí.

MEJORAS PREVIAS

Aquí detallaremos cada uno de las mejoras que efectuamos antes (algunas fueron mientras y otras después de la implementación del algoritmo, pero fueron planificadas con antelación) de la creación e implementación del algoritmo de administración de memoria heap en el kernel del sodium. Se especifica tanto la problemática como la solución.

Los ítems marcados con * fueron explícitamente pedidos por la cátedra.

- Existen dos segmentos de memoria para cada programa.

Problemática:

En nuestro trabajo práctico N°1, implementamos satisfactoriamente (sin saber que la cátedra requeriría el uso exclusivo de particiones), dos segmentos de memoria para cada proceso, pudiendo pasar parámetros al proceso sin "romper" el parte de la sección de código. Lo hicimos de manera satisfactoria, con soluciones temporales para los problemas de resolucion de CS y DS presentados en la compilación. El problema es que para implementar particiones debíamos volver al viejo esquema de 1 solo segmento de memoria por proceso.

Solución:

Debimos adaptar nuevamente al uso de un solo segmento todas las funciones del kernel involucradas (su mayoría dentro de gdt.c y syscall.c), la librería libsodium y el kernel shell para que vuelvan a funcionar con un solo segmento de memoria. También adaptamos todos los syscalls para que esto funcione.

Implicancias:

Perdimos la capacidad de pasarle parámetros al proceso sin alterar el code segment. Por el resto de las funcionalidades resultaron satisfactorias todas las pruebas efectuadas.

- Eliminar la excesiva cantidad de binarios presentes en la compilación

Problemática:

La cantidad de binarios sin uso aparente y con fines de pruebas (obsoletas) tienden a hacer mas pesada la distribución de sodium. Como ya sabemos, todos estos binarios se cargan en memoria baja (hasta 640KB) . Así que cuanto mas se acumulen mas probabiliades de pisar secciones del stack del kernel, además de librerías ya no usadas y secciones del código obsoletas. Además se genera code garbaging, que pensamos confundirá al alumnado del siguiente año, nuestra visión es mantener al sodium limpio y entendible.

Solución:

Decidimos adoptar medidas directas respecto a este problema, eliminando la mayoría de los binarios de prueba obsoletos, sus cabeceras, y sus entradas en los makefiles. Así como también buscamos código con FAN-IN 0 (nadie los llamaba, especialmente dentro del shell.c viejo) dentro del kernel buscando alivianar el tamaño de codigo del kernel, liberando espacio para el stack y

haciendolo mas estable. Obviamente lo hicimos efectuando pruebas y a consciencia de que algunas cosas son llamadas de forma no convencional como las interrupciones y callgates.

Implicancias:

Se ha reducido drásticamente el tamaño del sodium en disco y no han sucedido mas errores por tamaño de binarios en memoria como hemos llegado a tener en un punto.

- Eliminar los callgates adicionales

Problemática:

Una vez demostrado que puede navegarse a cualquiera de los 4 niveles de privilegio, solo quedan segmentos de código en memoria que ocupan espacio y entradas en le GDT. Todavía puede mantenerse el navegador en 3-0-3 como demostración de la funcionalidad de los callgates, pero es necesario remover la mayor cantidad de "basura" residual de pruebas y demostraciones.

Solución:

Estos callgates y sus entradas en GDT fueron eliminados.

Implicancias:

Ninguna.

- Mover los .h de Usuario a la carpeta include/usr

Problemática:

Por razones lógicas (estaban funcionando en nivel 0) algunos de los binarios compilados dentro de la carpeta /usr tenían su respectivo .h en la carpeta /kernel. Esto es un error conceptual ya que confundiría a los posibles nuevos desarrolladores de sodium.

Solución:

Pasamos todos los .h de usuario a la carpeta /include/usr y modificamos los includes necesarios para esto.

Implicancias:

Ninguna.

- Mover los .h de Usuario a la carpeta include/usr

Problemática:

Por razones lógicas (estaban funcionando en nivel 0) algunos de los binarios compilados dentro de la carpeta /usr tenían su respectivo .h en la carpeta /kernel. Esto es un error conceptual ya que confundiría a los posibles nuevos desarrolladores de sodium.

Solución:
Pasamos todos los .h de usuario a la carpeta /include/usr y modificamos los includes necesarios para esto.

Implicancias:
Ninguna.

- Problema con buffer de teclado*

Problemática:
En los comandos que requerían presionar una tecla para continuar, eventualmente luego de una segunda ejecución, no respetaban mas la espera y mostraban todo de una vez. Esto era debido a una mala implementación del leer_caracter en el macro de espera

Solución:
Se arregló la sintaxis de leer_caracter dentro del macro.

Implicancias:
Funcionan todos los comandos con espera de teclado ahora.

- Numeración de syscalls POSIX*

Problemática:
Nuestra numeración del syscall que utilizamos para los servicios kernel al proceso de shell usuario no respetaba (se solapaba) el estandar posix.

Solución:
Esto ha sido corregido asignandole un numero libre no usado por el estándar.

Implicancias:
Ninguna.

- Autocompletar no implementado*

Problemática:
Faltaba desarrollar la funcionalidad de autocompletar en el shell usuario.

Solución:
La funcionalidad fue desarrollada y responde con la tecla TAB al comando ingresado, conteniendo a todos los comandos presentes en el shell.

Implicancias:
Ninguna.

- Tecla F2 no recibida*

Problemática:

La tecla F2 no estaba siendo recibida por nuestro shell usuario.

Solución:

Esto fue arreglado manejando las otras entradas referidas a teclas especiales del teclado en el syscall de leer carácter.

Implicancias:

También se pueden recibir las flechas en el Log pudiendo navegarlo tanto para arriba como para abajo y también borrarlo.

- Hacer que el shell utilice espera pasiva.

Problemática:

Nuestro programa shell de usuario utilizaba un while(1) que efectuaba un el sycall read() tantas veces como podía hasta que recibía un carácter del teclado. Entonces lo procesaba y volvía a mantener la espera activa. Esto era poco eficiente y también poco vistoso, ya que por cada quantum de tiempo se cambiaba de proceso, y podía verse en la parte superior el cambio.

Solución:

Creamos una nueva entrada en la tabla de pcb del proceso que indica si éste esta esperando una entrada de teclado. El primer intento con read deja al proceso en estado ESPERANDO y activa este flag de espera. Luego, cuando se recibe una entrada por teclado, el manejador pregunta en orden ascendente en la tabla de pcb cuales procesos esperan la entrada de teclado. Al primero en esta tabla, se le pone en estado LISTO y se le baja este flag. Cuando retoma ejecución se encuentra con el código de tecla listo para ser usado.

Implicancias:

Durante el tiempo en que un proceso espera la entrada por teclado, ya no consume mas tiempo valioso de CPU y cambios de contexto de otros procesos que si tienen cosas que hacer mas que un while(1).

- Arreglar FORK*

Problemática:

En nuestra entrega final del TP1, el sodium implementaba satisfactoriamente el syscall fork. Dado que el fork utiliza posiciones de memoria muy particulares, cualquier cambio en el esquema de memoria lo deja no funcional. Esto mismo sucedió cuando cambiamos, entre otras cosas, el esquema de dos segmentos a una sola partición. El fork ya no funcionaba.

Solución:

Debimos revisar la estructura del fork y ver cuales fueron las implicancias de todos los cambios efectuados sobre este syscall. Por suerte la experiencia de haberlo hecho ya una vez nos dio la habilidad de solucionarlo en no mas de 4 horas. Gracias al debugger interno del bochs, pudimos hacer un seguimiento tanto del stack de kernel como el de usuario para saber qué valores se modifican en el nuevo PCB y TSS.

Implicancias:

Ahora ya puede usarse con completa libertad y hasta recursivamente el fork (solicitamos al usuario mesura respecto a esto, ya que el sodium no cuenta con chequeo de limites de stack y puede 'explotar' por consumo de memoria)

- Arreglar EXEC

Problemática:

Idem Fork pero decidimos hacerlo dado que queremos dejar al sodium lo mejor posible operativamente. Como es un sistema operativo didáctico, no hay mejor manera de ver como se ejecuta un nuevo proceso de manera posix que usando fork y exec y NO el atajo que utilizaba hasta ahora que era utilizando la funcion crear proceso directamente.

Solución:

El proceso fue el mismo que para Fork.

Implicancias:

Cualquier proceso ahora puede ser reemplazado por un .bin especificado. No importa si el tamaño de este es mayor o menor (de el tema tamaño de memoria será un tema recurrente en el transcurso de este TP y tiene que ver también con FORK y EXEC al ser los mecanismos principales para la creación de procesos y alocación de nuevos segmentos en la memoria alta del heap del kernel.).

- Tamaños de Stack (Kernel y Usuario) incorrectos

Problemática:

Los tamaños definidos de stack tanto para kernel como para usuario estaban definidos excesivamente cortos. Esto ocasionaba problemas de sobreescritura de código (problema de no tener varios segmentos). En usuario esto ocasionaba que las funciones del libsodium fallaran sin motivo, y en kernel, ocasionaba que algunos syscalls perdieran la información del proceso que los llamó.

Solución:

Se agrandó la constante que define el tamaño para stack de usuario. Ahora ya no ocurren mas incidentes como ese. En kernel se corrió el inicio del mismo y se achicó el heap. Además se corrió la asignación de 2 megabyes por parte del RamFS que se asignaba en memoria baja (¡ Esto es mayor que los 640kb disponibles) a memoria alta (de usuario). Por lo que ahora ya no hay problemas respecto a eso.

Implicancias:

El sodium entero tanto kernel como programas usuario se tornaron mucho mas estables y ya casi no hay incidentes de opcode inválido y otros errores 'impredecibles' que solían haber.

- Code Bombs

Problemática:

La problemática todavía no es muy clara. Al parecer faltas de programación y planificación ocasionan que ciertos datos se guarden en posiciones fijas (esto es una conjetura). Esto no habría causado problemas antes ya que el kernel era chico. Ahora esas posiciones estan ocupadas por código (cercano a syscall.c). Entonces cualquier funcion que este bajo esta "lluvia de bombas" se corrompe por completo. La posición de la función afectada varía en tanto se agregue o quite código detrás de ella.

Solución:

Una solución temporal para estas bombas es generar charcos de código inócuo para ser corrompido. Si un error impredecible sucede en una función, es probable que sea víctima de corrupción de este tipo, por lo que se intenta llenar ese espacio con código inócuo (muchos nops, quizás). De manera que este código reciba el daño y la función siga intacta.

Implicancias:

A pesar de que esta solución funciona bien, deberían encontrarse las posiciones exactas de daño y mas difícil aún, los causantes de esto. Queda como investigación a futuro.

- Falta de comentarios

Problemática:

Muchas de las funciones existentes y las nuevas que habíamos creado estaban sin cabecera compatible con doxygen y una descripción de su nombre, función, parámetros y valor de devoluión

Solución:

Decidimos, por lo menos, comentar y agregar las cabeceras pertinentes a las funciones agregadas por nosotros, de esta manera se complementa con este documento para la entendibilidad del sistema operativo para los siguiente cursos.

Implicancias:

Los profesores y la cátedra, y los alumnos que investiguen al sodium tendrán una herramienta mas de ataque para entender este TP y su implicancia sobre el aprendizaje de un sistema operativo.

ALGORITMO DE ADMINISTRACIÓN DE MEMORIA

Abarcaremos en esta sección primero las alternativas en materia de estrategias de administración de un heap que contemplamos antes de decidirnos por una. Todas éstas independientemente de toda implementación. Esta sección es puramente teórica y sus contenidos solo contienen lógica de manejo de heap, en la sección siguiente especificaremos en detalle como y en qué sentido fue implementado el algoritmo elegido.

Alternativas de implementación:

- Vector de punteros a memoria Heap

Descripción:

Esta estrategia consiste en utilizar un vector de tamaño limitado fuera o dentro de la zona de heap.compuesto por una estructura con tres campos:

- Flag Ocupado / Libre: Indica si el bloque apuntado está siendo utilizado o es un bloque disponible para ser reservado.
- Tamaño de Bloque: El tamaño en unidades del bloque apuntado
- Puntero al Bloque: Dirección en heap del bloque apuntado.

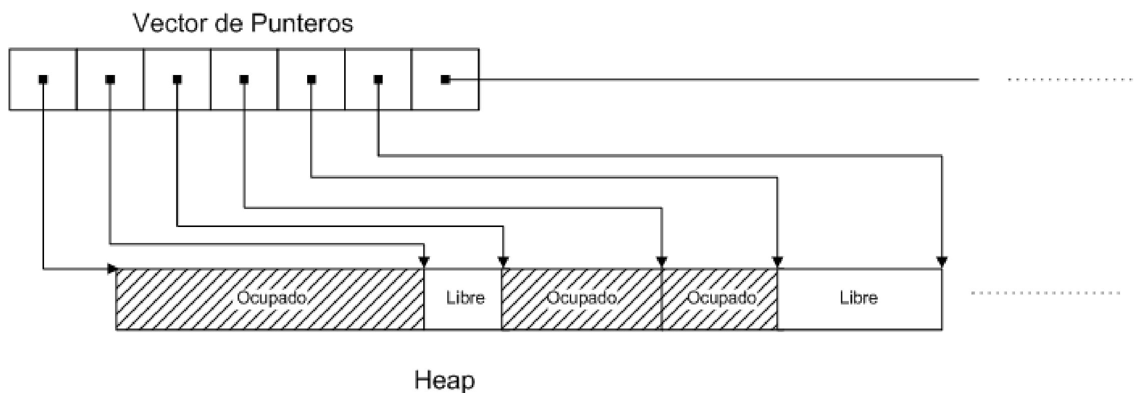

Figura 1: Vector de punteros a memoria

Estrategia Malloc:

Para la reserva de memoria se recorre secuencialmente el vector buscando alguno que apunte a un segmento libre en el heap que además cumpla la condición de estar libre. En el peor caso es necesario recorrer todo el vector hasta encontrar un puntero libre.

Estrategia Free:

Para liberar un segmento de memoria primero se busca a la posición que apunte a la dirección inicial del segmento a liberar y se cambia su estado de ocupado a libre. Luego se evalua el estado de los segmentos contiguos para saber si alguno está libre. Si el siguiente lo está, se elimina la entrada que apunta a ese y se aumenta el tamaño del actual con el tamaño del siguiente (merging). En tanto que si el anterior lo está, se hace el proceso similar pero adjuntando el recién liberado.

Ventajas:

Este algoritmo cuenta con la aparente ventaja de ser fácil de implementar pero es totalmente lo contrario por los siguientes motivos…

Desventajas:

- Es limitado. El tamño del vector debe ser fijo y predefinido. No pueden haber mayor cantidad de segmentos de memoria que la de entradas en el puntero
- Cada entrada ocupa mucha memoria. El hecho de no solo guardar el puntero, sino el tamaño y el estado es un desperdicio importante de memoria dado que otros algoritmos pueden reducir este gasto
- Su complejidad se vuelve inmanejable y los casos especiales son muchos, hay que practicar limpieza y reorganización del vector seguido cuando los punteros empiezan a cruzarse, la complejidad empieza a exigir recorrerlo varias veces.
- Al asignar una zona libre mayor a la solicitud se necesita un puntero extra para apuntar a la nueva zona libre además del bloque ocupado, por lo que este algoritmo es altamente sensible a la fragmentación.

- Punteros a bloque en Heap

Descripción:

Esta estrategia consiste en posicionar los punteros dentro del heap junto con los segmentos de información. Además es necesario un puntero en una posición fija que apunte al primero de los punteros en heap. La estructura de estos punteros contiene los siguientes campos:

- Flag Ocupado / Libre: Indica si el bloque apuntado está siendo utilizado o es un bloque disponible para ser reservado.
- Tamaño de Bloque: El tamaño en unidades del bloque apuntado
- Puntero al Bloque: Dirección en heap del bloque apuntado.
- Puntero al siguiente puntero: Se mantiene como una lista no ordenada de punteros.

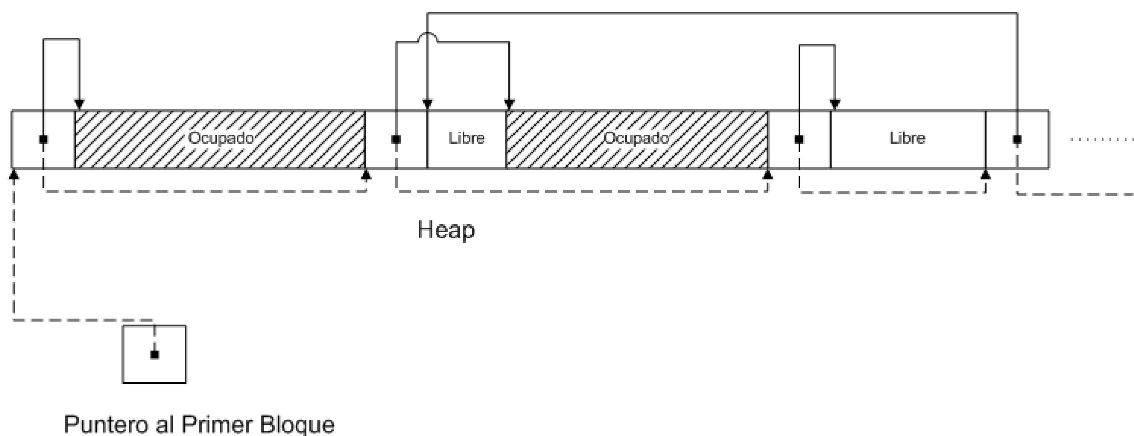

Figura 2: Punteros a bloque

Estrategia Malloc:

Se recorre la lista de punteros utilizando los punteros a los siguientes nodos hasta que se encuentra uno que apunta a una posición de memoria no ocupada y de tamaño igual o mayor al necesitado.

Estrategia Free:

Para liberar un segmento de memoria primero se busca a la posición que apunte a la dirección inicial del segmento a liberar y se cambia su estado de ocupado a libre. Luego se evalua el estado de los segmentos contiguos para saber si alguno está libre. Si el siguiente lo está, se elimina la entrada que apunta a ese y se aumenta el tamaño del actual con el tamaño del siguiente (merging). En tanto que si el anterior lo está, se hace el proceso similar pero adjuntando el recién liberado. El costo de esta operación es elevado teniendo en cuenta que por cada pregunta debe recorrerse una lista que puede contener muchos nodos ya que no se cuenta ahora con la restricción de un vector fijo.

Ventajas:

- Este algoritmo no cuenta con la limitación de cantidad de segmentos definible ya que se pueden definir indeterminada cantidad de punteros mientras alcance el tamaño del heap.
- Una posible mejora sería dividir la lista en dos. Una para bloques ocupados y otra para bloques libres. Esto a pesar de que aumentaría la complejidad del algoritmo, ahorraría procesamiento para encontrar bloques libres para el malloc.

Desventajas:

- Es demasiado complejo y como el anterior, cuenta con muchísimos casos particulares de conflictos propios del uso de listas. La cantidad de cálculos es mucha, más aún si se usan dos listas.
- Ocupa lugar del heap. El gasto es aún mayor con la utilización de dos punteros. Si la mayoría de las solicitudes serán de tamaños pequeños como 20 bytes y cada puntero ocupa 8 bytes, junto al resto de la información de nodo, se excedería en la mayoría de los casos el tamaño de meta-información al tamaño de memoria reservado.
- Al encontrarse dentro mismo del heap, es frágil ante errores de asignación. Si algun proceso excede su tamaño, podría dañar un puntero perdiendo toda la lista, lo mismo si es ocasionado por una falta de programación.

- Asignación contigua simple.

Descripción:

La información del Bloque de memoria se guarda intrínsecamente dentro del mismo. Con esta información básica es posible conocer la posición del siguiente bloque contiguo en memoria, por lo que no es necesario en absoluto la utilización de punteros. Cada bloque es autocontenido e independiente, y sus dos secciones: Meta-datos y Espacio utilizable puede ser visto como una unidad. Los metadatos son los siguientes:

- Flag Ocupado / Libre: Indica si el bloque apuntado está siendo utilizado o es un bloque disponible para ser reservado.
- Tamaño de Bloque: El tamaño en unidades del bloque apuntado

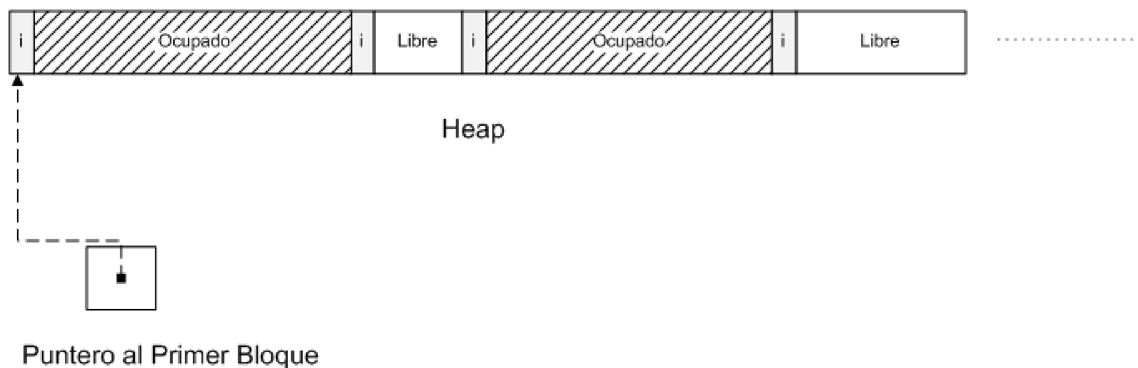

Figura 3: Asignación contigua simple

Estrategia Malloc:

Se recorre el heap 'saltando' de bloque en bloque utilizando la información en su cabecera. Sabiendo su posición inicial y su tamaño, es posible conocer la posición inicial del siguiente bloque en memoria. En cuanto se halla alguno libre y de tamaño suficiente, se lo marca como ocupado. La parte libre restante (si quedara) recibe una cabecera similar con el flag de ocupado en 0 y el tamaño que sobró. (El caso en que sobró espacio pero menor al tamaño de la cabecera lo trataremos en profundidad mas adelante)

Estrategia Free:

Para liberar un segmento de memoria primero se utiliza el mismo modo de búsqueda del malloc. Cuando se lo encuentra, se le marca el flag como libre y se efectua merging si es necesario.

Ventajas:

- No tiene ningun tipo de limitación respecto a cantidad de segmentos.
- La cantidad de espacio requerida para los metadatos es mínima. No hay necesidad de utilizar punteros.
- Es auto-ordenada. Esto significa que cada bloque siempre apunta al siguiente, esto simplifica al extremo la implementación ya que no ocurren asignaciones cruzadas como con el uso de punteros
- Es auto-compactada. Tiende a la compactación ya que se pregunta secuencialmente desde el principio en la asignación. Siempre va a tender a quedar un bloque libre al final.
- Es muy sencillo de programar e implementar.
- Proporciona soporte para particionado fijo y variable con una base común de lógica.

Desventajas:

- Es la mas frágil de las tres implementaciones. Cualquier exceso de parte del uso del bloque asignado o falla en la programación puede ocasionar una pérdida sucesiva de bloques. Para este fin decidimos hacer una implementación inicial externa al sodium que en las siguientes páginas comentaremos.

Detalles del algoritmo:

- Justificación de la elección.

Decidimos utilizar la asignación contigua simple debido a su simplicidad de programación y lo compacto de su tamaño. Si el tamaño se maneja con un tipo de dato long int (4 bytes en C) y el estado Libre/Ocupado con un unsigned char (1 byte), se estan ocupando solo 5 bytes de memoria en heap por cada bloque. Podemos considerar que este es un tamaño relativamente chico comparado al de los demas algoritmos descriptos (y al algoritmo que usaba anteriormente el sodium).

Nuestra visión de usar este algoritmo es, por un lado, la practicidad y la simplicidad. El código es entendible, elegante y limpio, el concepto es fácil de aprender por quienes podrían analizarlo. Consecuencia de esto es la robustez. Un algoritmo de concepto simple y con programación cuidadosa tiende a tener muchas menos fallas. Debido a que en un comienzo supimos que todo el desarrollo iba a depender del algoritmo, decidimos crearlo en un entorno libre de contaminación en cuanto a la implementación.

Este algoritmo fue desarrollado y testeado infructuosamente en lenguaje C, y en el sistema operativo Windows, utilizando una salida a pantalla que mostrara el estado de la misma y que permitiera efectuar mallocs y frees y observar su comportamiento. De esta manera intuitiva logramos depurar el algoritmo de manera de asegurarnos que no produzca errores impredecibles en un futuro.

Estas son algunas imágenes del algoritmo desarrollado en windows:

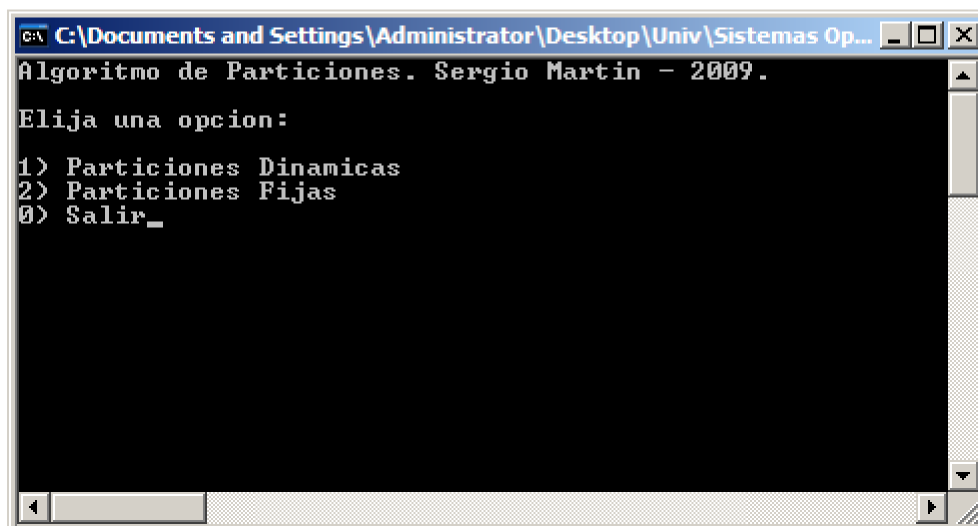

Figura 4: Menu de seleccion de tipo de partición

El Menu permite elegir entre dos tipos de particionado: Variable y Fijo. El funcionamiento de cada uno se tratará luego.

```
C:\Documents and Settings\Administrator\Desktop\Univ\Sistemas Operativos\TP3\mem_part.exe
Algoritmo de Particiones. Sergio Martin - 2009.   Modo: Variable.

Total: 3500

Elija una opcion:

1) Malloc
2) Free
0) Salir
```

Figura 5: Estado Inicial del modo variable

El estado inicial de la pantalla muestra un heap vacío salvo por algunos símbolos, a continuación se detallan las diferencias tipo de Particion:

- Fijo: Se setean todos los bloques de tamaño fijo como libres desde un comienzo. Luego se asigna o desasignan manteniendo el tamanio. No hay merging de espacios libres.
  Es necesario achicar el heap hasta un numero multiplo del tamanio del bloque.

- Variable: Se asigna al principio un solo bloque libre conteniendo todo el heap. Luego se iran asignando o desasignando pudiendo variar el tamaño y la cantidad de bloques.

La nomenclatura de lo mostrado por pantalla es la siguiente:

Cada uno de los simbolos que se muestran en pantalla representan 1 byte y cada uno tiene un significado propio:

"C" - Es una representación de la parte de codigo, datos y stack del proceso antes de comenzar el heap.

"I" - Son la cabecera de cada bloque de datos. Ocupan espacio y no guardan mas que la información de cada bloque.

"(#)" - Representan lo mismo que I pero sirven para indicar el numero de ID del bloque. Puede interpretarse igual que I.

"S" - Indica lo mismo que I pero sirve para indicar que se trata de la cabecera de un bloque libre.

"-" - Es el contenido de los bloques libres.

"#" - Es el contenido de los bloques usados. Al igual que "-" sirven para representar el espacio ocupado(libre) por el bloque disponible para datos en sí y no de cabecera.
El tamaño total del bloque debe interpretarse como la suma entre su cabecera y sus bytes de datos, es decir: cantidad de I + cantidad de # para bloques usados, o cantidad de S + cantidad de - para bloques libres.

Figura 6: Se reservó un bloque de 200 bytes

Se puede apreciar en la figura 6 que se agregó un bloque de 200 bytes. Al hacer esto se generó una cabecera para el bloque asignado, y una nueva cabecera (representada por S) que indica el comienzo del nuevo segmento de espacio libre.

```
C:\Documents and Settings\Administrator\Desktop\Univ\Sistemas Operativos\TP3\mem_part.exe
Algoritmo de Particiones. Sergio Martin - 2009.   Modo: Variable.

Total: 3500

Elija una opcion:

1) Malloc
2) Free
0) Salir_
```

Figura 7: Se agregaron 2 bloques mas.

En la figura se puede apreciar que se han agregado dos bloques mas. Ahora mostramos el ejemplo de un free para el bloque de ID = 2.

```
C:\Documents and Settings\Administrator\Desktop\Univ\Sistemas Operativos\TP3\mem_part.exe
Algoritmo de Particiones. Sergio Martin - 2009.   Modo: Variable.

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC<1>IIIIIIIII####...

Total: 3500

Elija una opcion:

1> Malloc
2> Free
0> Salir_
```

Figura 8: Se liberó el bloque (2)

Se puede observar en la figura 8 como el espacio otrora ocupado por el bloque (2) ya está libre para su uso. De este gráfico surge la pregunta: ¿Qué pasaría si libero (1) o (3)? ¿Quedan dos bloques libres contiguos? La respuesta es no. El algoritmo utiliza el merging para bloques de espacio libres contiguos.

El merging consiste en preguntar después de cada ejecución de un free si existen espacios libres contiguos ya sea a izquierda y/o a derecha (antes o después) del bloque a liberar. De ser así ambos bloques se unen hacia izquierda, es decir, el de la izquierda se 'come' al de la derecha quedandose con tu tamaño y quedando una sola información de cabecera.

En el caso de que el bloque a liberar tenga bloques libres a izquierda y a derecha, primero el bloque central se 'come' al de la derecha y luego el de la izquierda se 'come' al central (que ahora es mas grande) quedando un solo bloque libre del tamaño de la suma de los tres y que empieza donde empezaba el de la izquierda.

A continuación mostramos un ejemplo gráfico de merging.

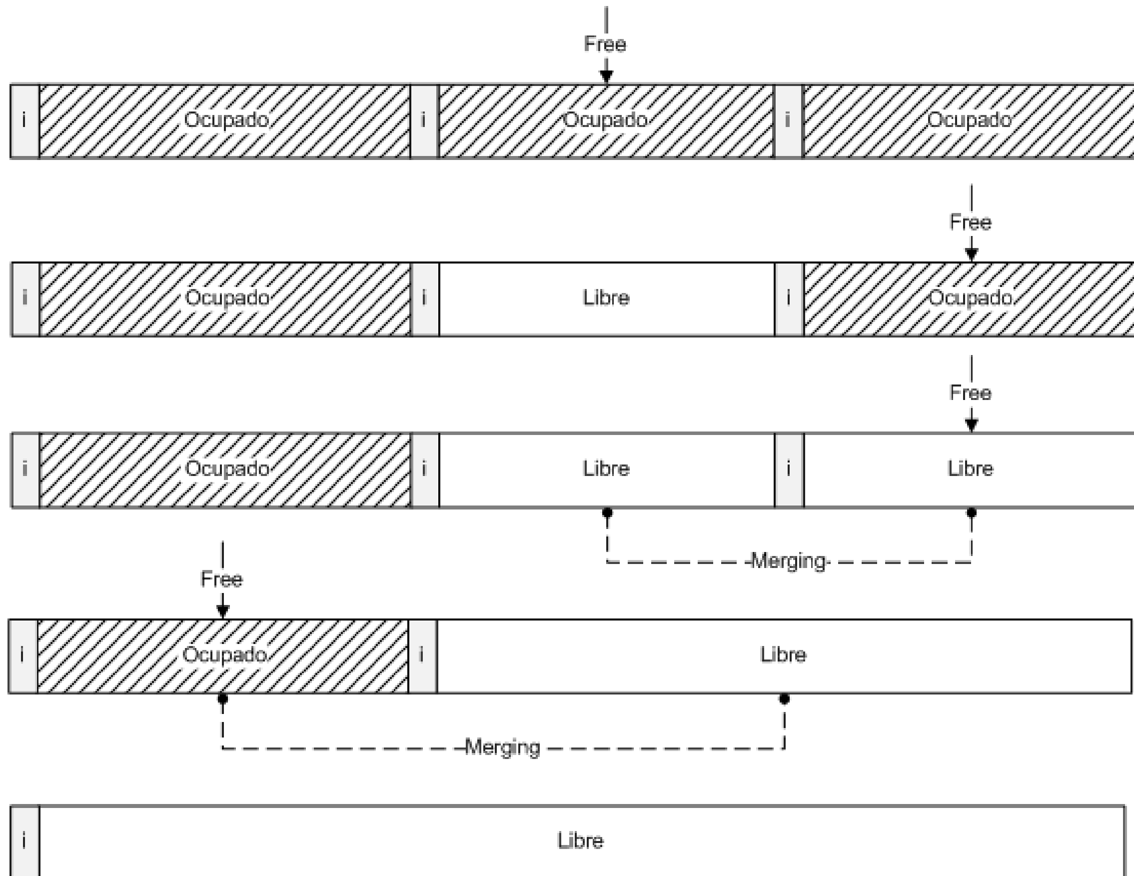

Figura 9: Ejemplo de merging a izquierda y a derecha

En la figura 9 puede observarse las consecuencias de los diferentes free(). El primero causa que el bloque central se libere, luego pregunta si alguno de los dos contiguos esta libre también. Al ser esto negativo termina ahí.

Luego para el free del bloque de la derecha, detecta que el bloque central está libre así que efectua un merging entre ambos. Lo mismo pasa luego pero para el bloque de la izquierda que detecta un bloque libre a derecha.

Vemos en la siguiente imagen del algoritmo en ejecución que sucede exactamente esto cuando liberamos el bloque (1):

```
C:\Documents and Settings\Administrator\Desktop\Univ\Sistemas Operativos\TP3\mem_part.exe
Algoritmo de Particiones. Sergio Martin - 2009.   Modo: Variable.

Total: 3500

Elija una opcion:

1) Malloc
2) Free
0) Salir
```

Figura 10: Ejemplo de merging en nuestro algoritmo

Puede observarse que en vez de quedar dos bloques libres antes de (3), ambos bloques libres se unen para formar un solo y del tamaño de la suma de ambos.

Figura 11: Particiones fijas

En la figura 11 puede verse un ejemplo de particiones fijas y cómo se puede producir fragmentación debido al tamaño fijo de la partición. Las particiones fijas son en nuestro algoritmo un caso particular de las variables donde:

- No hay merging de espacios libres
- Se asignan un tamaño fijo siempre ante mallocs incluso menores

La razón para que exista fragmentación es la siguiente:

Siendo el tamaño de partición fija de 220 unidades y el tamaño total 3450 (en realidad 3500, pero 50 son pensados para el código) unidades, la cantidad de particiones posible es la división entera entre ambos = trunc(3450/220).
En este caso el resultado es 15 bloques de 220 y una fragmentación (espacio sin usar) resultante de 150 unidades desperdiciados.

Invitamos a la cátedra a solicitarnos este programa y examinar todas sus funcionalidades, ya que sirve como guía visual para aprender fácilmente los conceptos de partición fija, variable y las diferentes situaciones en las que puede incurrir incluída la fragmentación que ocurre al utilizar partición fija (queda un espacio libre pero que no alcanza a satisfacer el tamaño del bloque fijo), o la fragmentación interna de la partición variable que ocurre cuando el tamaño libre restante de un bloque es menor al tamaño de cabecera (en nuestro algoritmo este espacio sobrante se une al bloque para evitar conflictos).

IMPLEMENTACIÓN

Introducción:

La implementación de nuestro algoritmo en sodium presenta muchas aristas ya sea lógicas, estructurales, estratégicas y de adaptación. Comenzaremos por describir el modelo estructural y lógico ya casi nisiquiera relacionado al sodium en sí sino al utilización de este o cualquier otro algoritmo de administración de heap tanto en el kernel de un sistema operativo como en el heap de un proceso usuario.

La premisa básica ya descripta en el capitulo anterior es mantener la simplicidad y la elegancia tanto en la codificación como en la implementación. Por lo que no solo nuestro algoritmo presenta una base lógica identica tanto para particiones fijas como variables, siendo las primeras un caso particular de estas últimas, sino que también pueda ser indistintamente utilizable para cualquier tipo de heap con una mínima información básica.

A nivel implementación en un sistema operativo buscamos la misma capacidad de reutilización para poder administrar un heap indistintamente de si sea la memoria baja o alta del heap de kernel o la memoria heap de usuario.

<u>- Variables del algoritmo</u>

Para estos fines el algoritmo debe contar con algunos datos básicos para su funcionamiento.

- Posición Inicial del heap: Como el algoritmo está pensado para trabajar en entornos donde la posición inicial del heap no es 0, sino que hay un espacio de código a respetar, debe indicarsele la posición inicial a partir de la 0 en la que se encuentra el heap.

- Tamaño del heap: El algoritmo debe conocer el tamaño total en bytes del heap que va a administrar. (Esto es de particular importancia en particiones fijas)

- Tipo de particionado: Se debe indicarle que tipo de partición se utilizaran. Para esto hay 3 opciones:
    * Partición Variable
    * Partición Fija Equitativa*
    * Partición Fija Proporcional

  Quien esto escribe asume que el lector conoce perfectamente los tres conceptos y sus diferencias.

  *Para Partición Fija Equitativa, se debe proporcionar también el tamaño de partición.

Existen también variables particulares necesarias para procesos usuario, y también particulares para kernel. Pero de esto nos encargaremos mas tarde cuando describamos la implementación de cada uno en detalle.

- Estructura en memoria del SODIUM

Mostraremos a continuación un pantallazo de cómo está distribuida la memoria en SODIUM (y quizás de igual manera en muchos otros sistemas operativos).

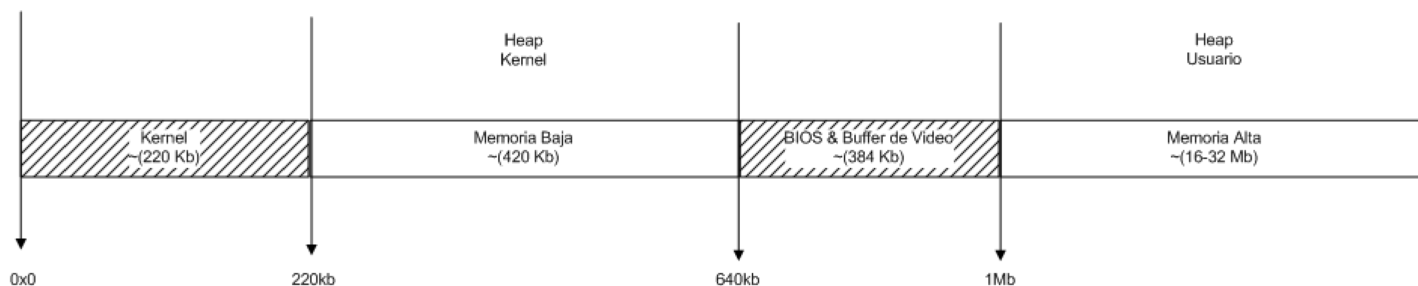

Figura 12: Distribución de la memoria

En la figura 12 observamos que la memoria normal está divida en varios segmentos (esta es una visión simplificada ya que intervienen también algunos elementos menores como stack de kernel que no viene al caso graficar en este punto).

Como primer segmento desde la posición 0, está situado el kernel del Sodium que ocupa aproximadamente 220 kilobytes. Desde la posición 220kb está el segmento de memoria libre baja que en el caso del kernel de sodium ocupa 420 kb. Luego de esta desde los 640kb y hasta el 1er MB, está la memoria reservada para Rutinas de BIOS y buffer de video. Esta última sección de memoria no es usable.

Por lo tanto si el kernel no puede ser tocado y la memoria de video esta protegida, nos quedan dos segmentos libres para utilizar como heap por parte del kernel que estan separados. En un algoritmo basado enteramente en la contiguo de sus bloques, esto a primera vista parece ser un problema. Sin embargo la solución es bastante simple.

Debido a la naturaleza flexible del algoritmo, podemos inicializar varios heaps en un mismo espacio de memoria no necesariamente contiguos. Basta con setear las variables básicas ya citadas con los datos de un heap, y ejecutar la función InicializarHeap(). Esta función (en el caso de partición variable) genera una cabecera con información de bloque libre y del tamaño del heap en la posición indicada. Con un puntero a esa dirección, ya tenemos generado un heap nuevo listo para ser usado.

Esta operación puede ser realizada tantas veces como heaps se quiera crear.
Podemos ver en la figura 12, que hay indicados 2 heaps sobre cada segmento de memoria disponible.

- **Heap de Kernel:** Contrario a DOS, en este heap decidimos llamarlo de "kernel" porque en el solo vamos a reservar espacios de memoria exclusivo para operaciones internas del kernel y nunca para programas usuario. De esta manera restringimos al máximo la utilización de esta área de memoria tán sensible (Recordemos que el sodium no es ninguna panacea y que es preferible utilizar segmentos de memoria lo mas alejados posible del kernel y el stack de kernel).

- **Heap de Usuario:** Decidimos llamarlo así no porque sea *administrado* por los procesos de usuario sino porque en esta parte de la memoria vamos a asignarle memoria a los procesos usuario (también al proceso reloj y shell, así como a la RAM FS que ocupa 2 MB y que no entra en memoria baja). De esta manera separamos lo suficiente los espacios de memoria de kernel de los de usuario.

Implementamos el algoritmo de manera que el sodium es capaz de manejar ambos heaps de manera simultánea e indistintamente ambos heaps. Para esto creamos una interfaz entre el kernel y el algoritmo. El siguiente esquema representa de manera básica la estructura de implementación:

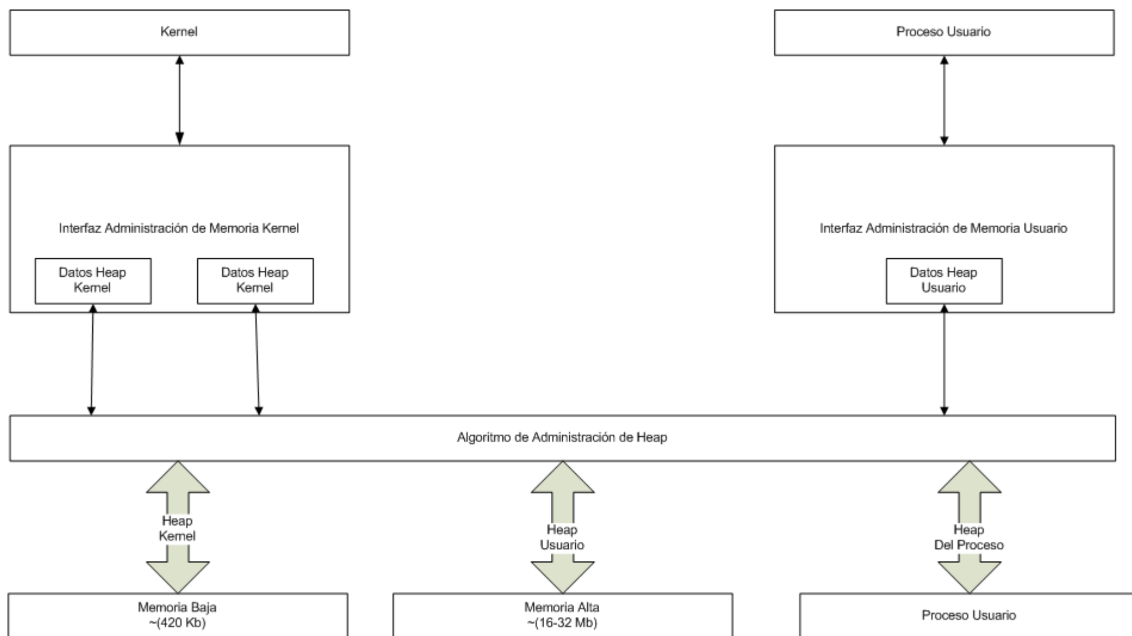

Figura 13: Estructura básica de heaps en SODIUM

- Implementación del algoritmo en Kernel.

Como ya explicamos, en kernel vamos a manejar 2 heaps. A continuación especificaremos cuando, como y qué se encarga de inicializarlos y utilizarlos.

Como Indica la figura 13, hay una interfaz para kernel del algoritmo de administración de memoria, que maneja las peticiones del mismo y configura el algoritmo base cada vez para poder interpretarlas.

La librería a ser incluída para la administración de heap es *mem_part_kernel.h,* que incluye todas las funciones de manejo de ambos heaps, tanto de memoria alta como baja. A su vez se debe incluír la librería del algoritmo base llamada *mem_part_firstfit.h* llamada así debido a que el algoritmo para la búsqueda y asignación de segmentos elige la primer opción válida. Queda para desarrollos posteriores del alumnado el desarrollo (en base a este) de nuevos tipos de algoritmo para la reserva de segmentos del heap.

La librería de interfaz de kernel presenta los siguientes servicios o funciones:

void vFnInicializarHeapsKernel();
void vFnConfigurarParticiones();
*void * pvFnReservarParticionUsuario(unsigned int);*
*void * pvFnReservarParticionKernel(unsigned int);*
*void * pvFnRedimensionarSegmento(void*, unsigned long);*
*void vFnLiberarParticion(void *);*

A continuación haremos una breve descripción de cada una.

- *vFnInicializarHeapsKernel():* Sirve para inicializar un heap. Una vez que estan seteadas las variables básicas para el algoritmo, se ejecuta esta función y se crea el primer bloque de memoria para ese heap en caso de ser particiones variables, o todos los bloques si es partición fija ya sea equitativa o proporcional

- *vFnConfigurarParticiones():* Se utiliza una única vez en el booteo del sodium antes de cederle el control al shell (ver main.c). Muestra un menú como se ve a continuación. Solo se ejecuta cuando se selecciona particiones fijas. Sirve para setear el tamaño en bytes de las particiones en memoria baja, y en porcentaje (para fijo solo se indica un solo tamaño) por cada uno de los bloques (para proporcional).
  Luego muestra una lista de todos tamaños finales de las particiones de memoria y sus respectivos porcentajes.

-

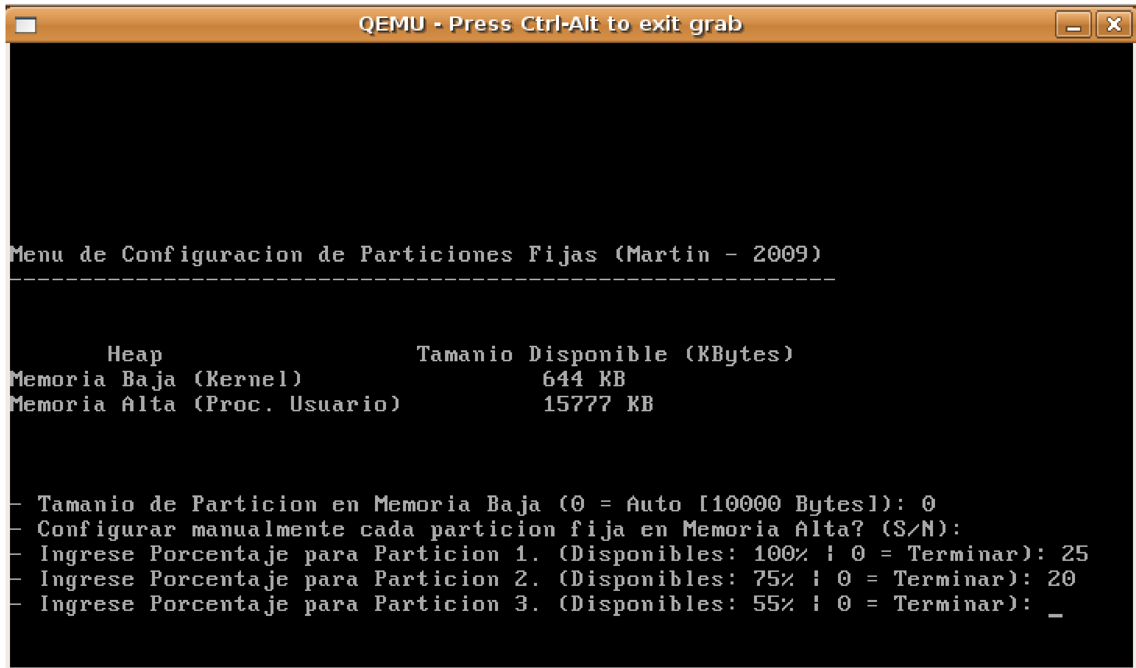

Figura 14: Menu de configuración de particiones

- *void * pvFnReservarParticionUsuario(unsigned int):* Reserva un segmento de memoria del tamaño pasado por parámetro en el heap de memoria alta (usuario) y devuelve la posición (posición inicial + tamaño de metadatos, devuelve la primera posición útil, no la primera posición base) del mismo.

- void * pvFnReservarParticionKernel(unsigned int): Reserva un segmento de memoria del tamaño pasado por parámetro en el heap de memoria baja (kernel)

- *void * pvFnRedimensionarSegmento(void*, unsigned long):* Utilizado desde la función vFnRedimensionarProceso de gdt.c para las solicitudes brk y sbrk de los procesos usuario (mas tarde abordaremos la cuestión del algoritmo en procesos usuario). Recibe como parámetro la dirección inicial del segmento y el tamaño nuevo deseado.

- *void vFnLiberarParticion(void *):* Sirve para Liberar un segmento de memoria. Es indiferente de qué heap sea, esta función llama al free del algoritmo pasandole la memoria física y cuando reconoce la cabecera de metadatos pone el flag de usado en 0.

Si el lector incurre en el analisis de estas funciones en el .c correspondiente, encontrará que la mayoría no hace mas que elegir el heap destino y utilizar la librería base del algoritmo, de manera que la implementación está altamente cohesionada.

Cabe notar también que tanto la interfaz de kernel como el algoritmo base se compilan e incluyen al kernel. El hecho de que las interfaces procesos usuario y de kernel recurran a la misma biblioteca base es desde el punto únicamente lógico, dado que el sodium maneja bibliotecas estáticas, este algoritmo debe compilarse y adjuntarse a ambos.

- Cola de espera de procesos

Como la Figura 14 lo muestra, es posible configurar manualmente el tamaño en porcentaje de cada partición fija. También es posible no utilizar toda la memoria disponible. El usuario en cualquier momento puede ingresar 0 deteniendo el ingreso de nuevos bloques de memoria. Esto deja una limitada cantidad de particiones disponibles para asignarle a nuevos potenciales procesos.

¿Qué pasaría si ya no quedan particiones libres y queremos ejecutar un fork, un exec (el exec requiere un segmento nuevo antes de desechar el viejo), o directamente usar la función crear_proceso del gdt.c?

La respuesta más rápida es que se reportaría falta de memoria y se daría de baja la petición. Pero también puede manejarse una cola de procesos en espera, como solicitó la cátedra.

En nuestra implementación utilizamos dos enfoques diferentes:

- Para creación de procesos Posix
- Para creación directa de procesos.

La creación de procesos POSIX experimenta dos etapas. Primero el proceso padre ejecuta un fork creando un proceso hijo. Este fork está condicionado de manera que el padre y el hijo ejecuten dos acciones diferentes. El padre ejecuta un waitpid(pid_hijo), esperando la señal SIGCHLD de que el hijo terminó su ejecución, y el hijo ejecuta exec("BINARIO.BIN"), reservando una nueva dirección de memoria .

La creación directa de procesos es utilizando crear_proceso de gdt.c y consiste en llamar la función especificando el binario a ejecutar. El kernel detiene al llamador, se crea un segmento de memoria y se ejecuta el proceso. Cuando termina, el kernel devuelve el control al llamador.

La primera opción es visible que para no dar de baja la petición de fork o exec, sería recomendable no una lista de espera (que sería impracticable debido a que un fork ahora no es lo mismo que un fork después, o cuando se pueda) sino un flag en su cabecera PCB que indique que esta en espera y bloquearlo. Lo mismo para exec.

Las solicitudes de crear_proceso si se manejan con una cola de espera de procesos, debido a que pueden continuar su funcionamiento y esos binarios luego pueden instanciarse.

La siguiente figura grafica la secuencia de acciones según qué haya solicitado el proceso.

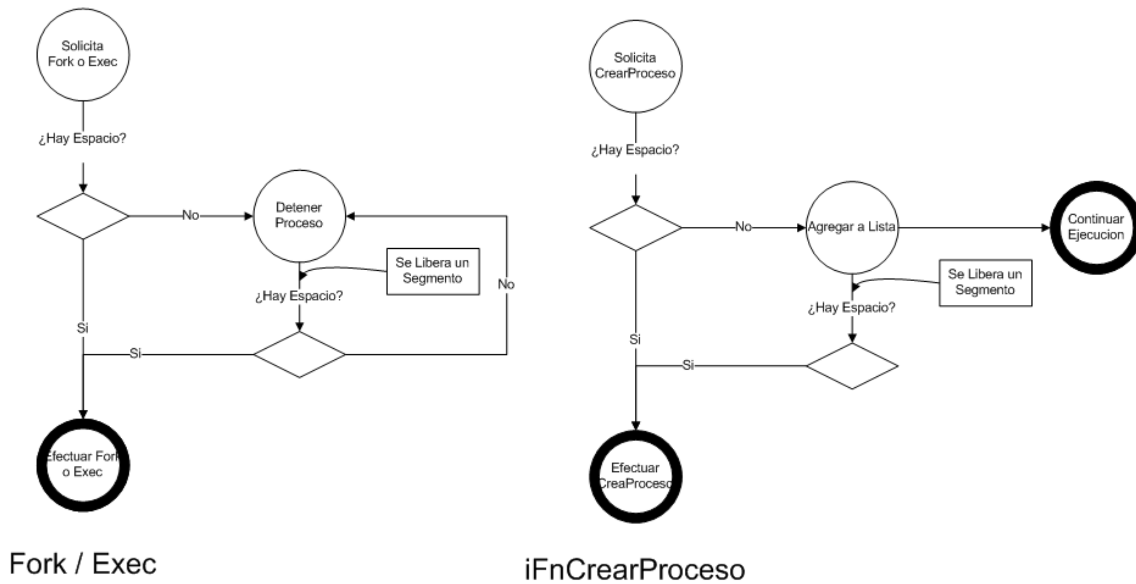

Figura 15: Modo de ejecución Posix / No Posix

Vamos a ver ahora un ejemplo funcional en sodium de un proceso en lista de espera.

```
QEMU
CSW: PID 1      TAREA=PR_Reloj....      INDICE GDT: 7        24/11/09 04:34:03

Iniciando reloj del sistema ...                          [ HECHO ]
Iniciando Proceso Reloj ...                              [ HECHO ]
Iniciando Task Register ...                              [ HECHO ]
Memoria total del sistema: 15 MB
Habilitando Interrupciones ...                           [ HECHO ]
Iniciando Proceso Shell ...                              [ HECHO ]
Se cargo la distribucion de teclado "Ingles US"
Cmd>listpart
Listado de particiones en heap de memoria alta.
-----------------------------------------------
Numero de Particion            Tamanio (KB)             Libre/Usado
             1                 1540                    Usado
             2                 1540                    Usado
             3                 1540                    Usado
             4                 1540                    Usado
Cmd>_

| ayuda | ps  |  init | mem | segs  | cls  | F2: Log |
```

Figura 16: Lista de particiones fijas

Utilizando el comando >listpart podemos ver (si utilizamos particiones fijas) la lista de particiones que tenemos libres/usadas y su tamaño. (Mas info sobre este y otros comando nuevos utilizando el comando >ayuda)

Podemos ver en la figura 16 que las 4 particiones fijas estan en uso. Sin embargo queremos ejecutar el proceso usuario y no nos importa esperar a que se libere alguna partición. Comencemos por utilizar la sintaxis >ejecutar usuario

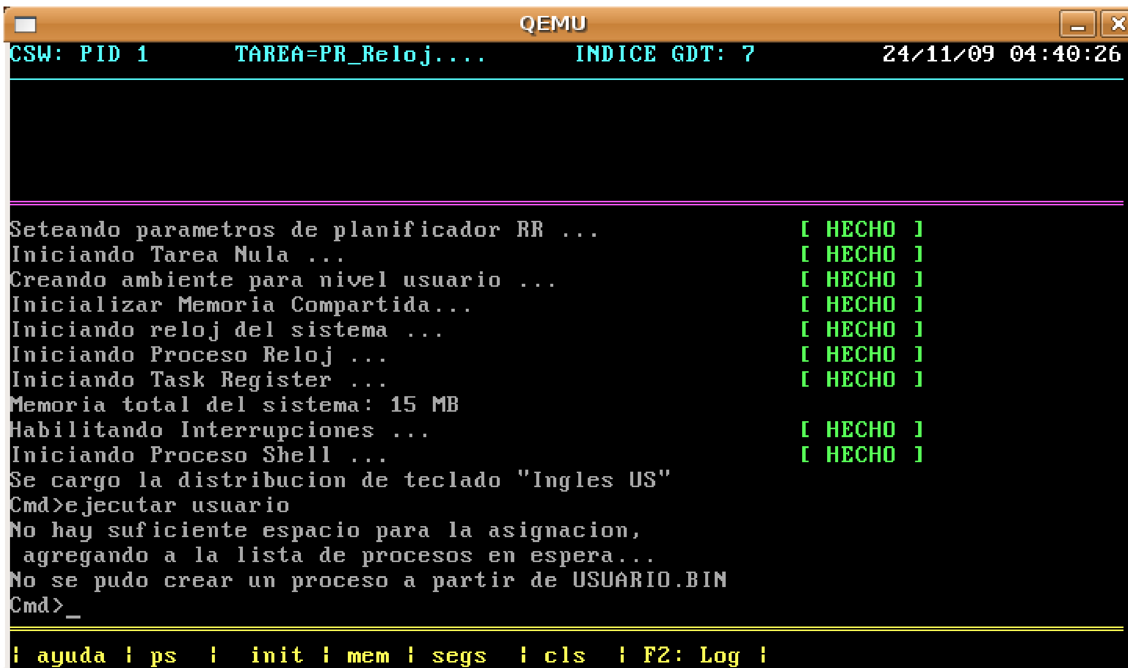

Figura 17: Proceso usuario agregado a lista de espera

Sodium indica que ya no hay particiones libres para cargar a USUARIO.BIN y se lo guarda en la lista de espera. A continuación mostramos la lista de espera junto con algunos procesos mas:

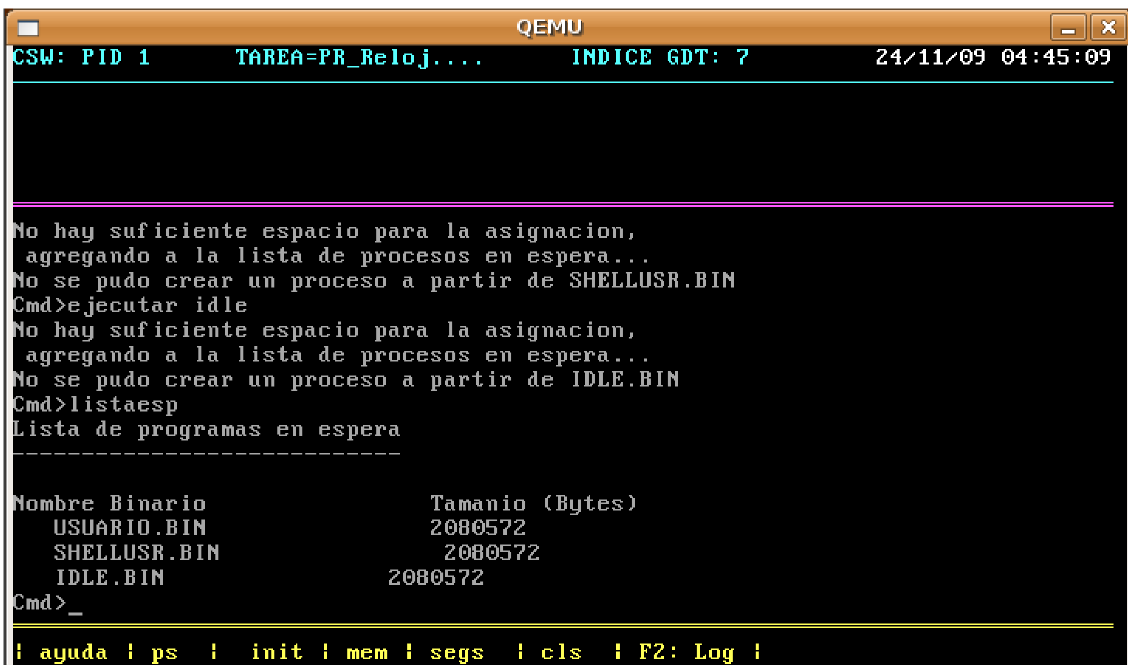

Figura 18: Lista de espera de procesos

De la misma manera cuando queremos ejecutar un Fork ante falta de memoria, el proceso se bloquea y aparece un mensaje indicandolo:

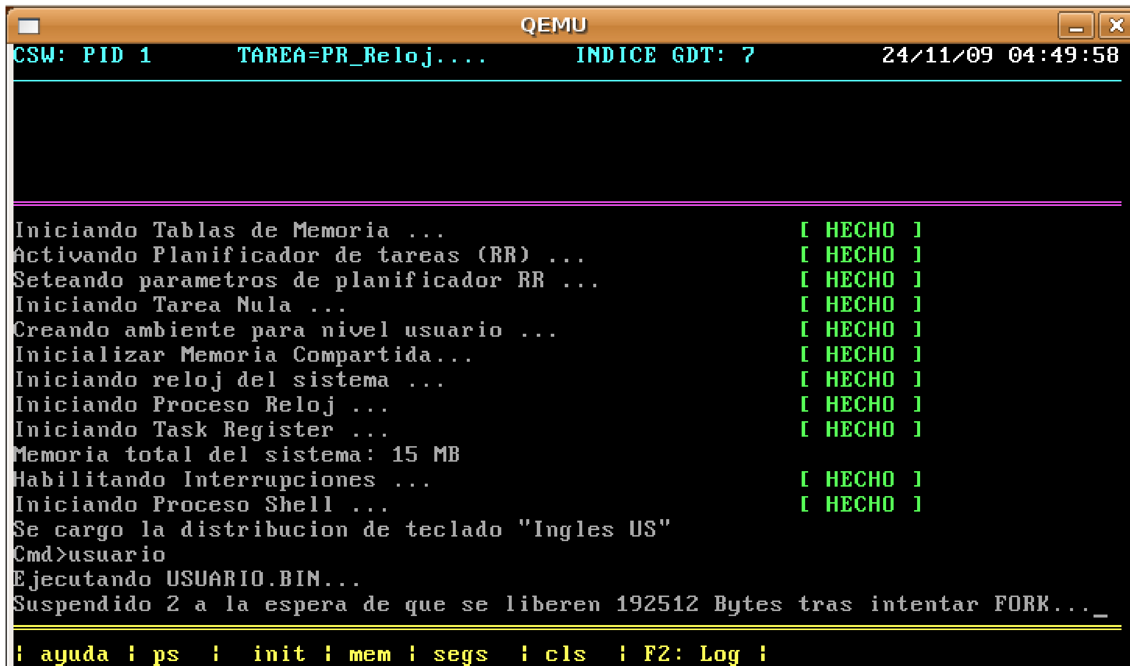

Figura 19: Fork suspendido hasta liberacion de memoria

Notese que la sintaxis para ejecutar un proceso de forma posix es llamando directamente el nombre sin el .bin, y para llamarlo usando iFnCrearProceso es con >ejecutar proceso.

- Implementación del algoritmo en Procesos Usuario

El proceso usuario contiene un solo heap situado luego de la sección de código, datos no inicializados, datos inicializados y sección de stack. Desde este punto hasta el final del proceso se situa el heap. El tamaño del mismo, y otras variables que configuran al administrador del heap se definen como variable de entorno desde el shell (esto lo detallaremos mas adelante).

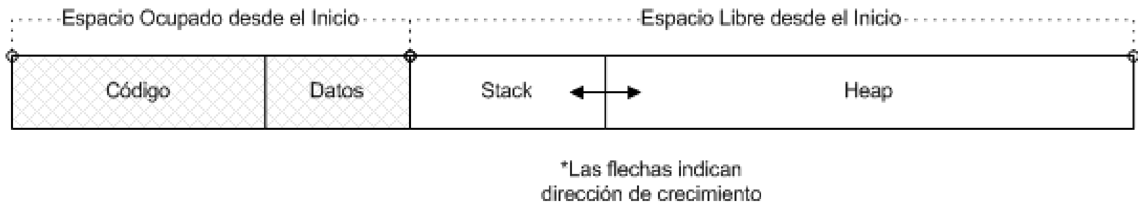

Figura 20: Estructura en memoria de un proceso

En la figura 20 se observa que hay un espacio reservado para la administración del kernel usuario. Este puede ser utilizado a travez de funciones por las funciones por defecto (malloc, free, realloc, calloc…). El administrador cuenta con un solo puntero no visible al usuario donde guarda la dirección inicial del heap, desde el cual puede acceder al resto de los segmentos utilizando el algoritmo indicado en la descripción del mismo.

Estas funciones básicas estan proveídas por la interfaz de usuario de nuestro algoritmo de administración de heap. Nuevamente, el algoritmo es el mismo que utilizamos para el kernel, y las funciones básicas de reserva y liberación de segmentos son en escencia iguales, pero existen algunas diferencias (a continuación las especificaremos).

Todas los servicios proveídos a los procesos usuario se encuentran en la librería mem_part_user.h ubicada en la carpeta /include/usr. Mostraremos una lista de funciones disponibles y luego detallaremos su funcionamiento.

void malloc(long int);*
void realloc(long int, unsigned long);*
int free(long int);
void iFnListar();
int iFnInicializarHeapUsuario();

- *void* malloc(long int):* Esta es la función básica con la que se reserva memoria en el heap. Se le especifíca el tamaño en bytes del bloque a reservar y devuelve la posición de memoria desde desde el punto cero del proceso (este cero es la dirección base vista por el propio proceso, pero no es el cero físico de memoria, por lo que la dirección física real resultaría de la suma entre la posición inicial del proceso mas el offset devuelto por la función.)

Malloc pregunta siempre antes de asignar si es que se ha inicializado el heap. La inicialización del heap no sucede al iniciar el proceso ya que esto sería un paso innecesario para los procesos que no lo utilizan y ademas sería difícil setear en creación este puntero. Si no ha sido inicializado, malloc llama a la función *iFnInicializarHeapUsuario.*

Si fue inicializada, llama a la función de la biblioteca propia del algoritmo que recorre el heap apuntado por el puntero a inicio y devuelve el primer segmento que de con las condiciones de reserva.

Si se queda sin espacio de heap, intentaremos solicitarle mas espacio al kernel. (syscalls brk o sbrk, solo en modo variable) En el caso de que el error en particiones fijas haya sido que el espacio solicitado es mayor al tamaño de la particion, esto se evalua hace por error codes (la lista de error codes la indicaremos luego).

- *iFnInicializarHeapUsuario():* Se utiliza una única vez luego del primer malloc para inicializar el heap del proceso. Obtiene los datos de configuración de la syscall no posix headerinfo pasandole la dirección a una estructura de la siguiente forma:

typedef struct {
  int iTipoParticion; // Indica si está en uso o es un bloque libre
  int iModoPartFija; // Indica que tipo de administracion si un tamanio fijo o basado en lista de porcentajes
  long int liNumeroTamPartFijaUser; // porcentaje para particion fija
    int iComienzoHeap; // Indica el comienzo del heap luego del codigo y el stack
    int iTamanioHeap; // El tamanio del heap lo determina una variable de entorno
    int piPorcentajesPartFija[100]; // Lista de porcentajes para particiones fijas variables
} t_Header;

La mayoría de estas variables se obtienen de variables de entorno definidas por el usuario utilizando el comando set. (detallaremos estas variables mas adelante).

Luego se recibir los datos de configuración inicializa el heap creando el primer nodo en la primer posición con el tamaño del mismo si es modo variable, o creando diferentes nodos fijos si se maneja partición fija. Esto lo hace llamando al inicializador del algoritmo base al que recurren tanto el kernel como el usuario.

- *void* realloc(long int, unsigned long):* Esta function recibe la dirección de memoria de un segmento reservado solicitando que se agrande su tamaño pero manteniendo los datos que hay en él.

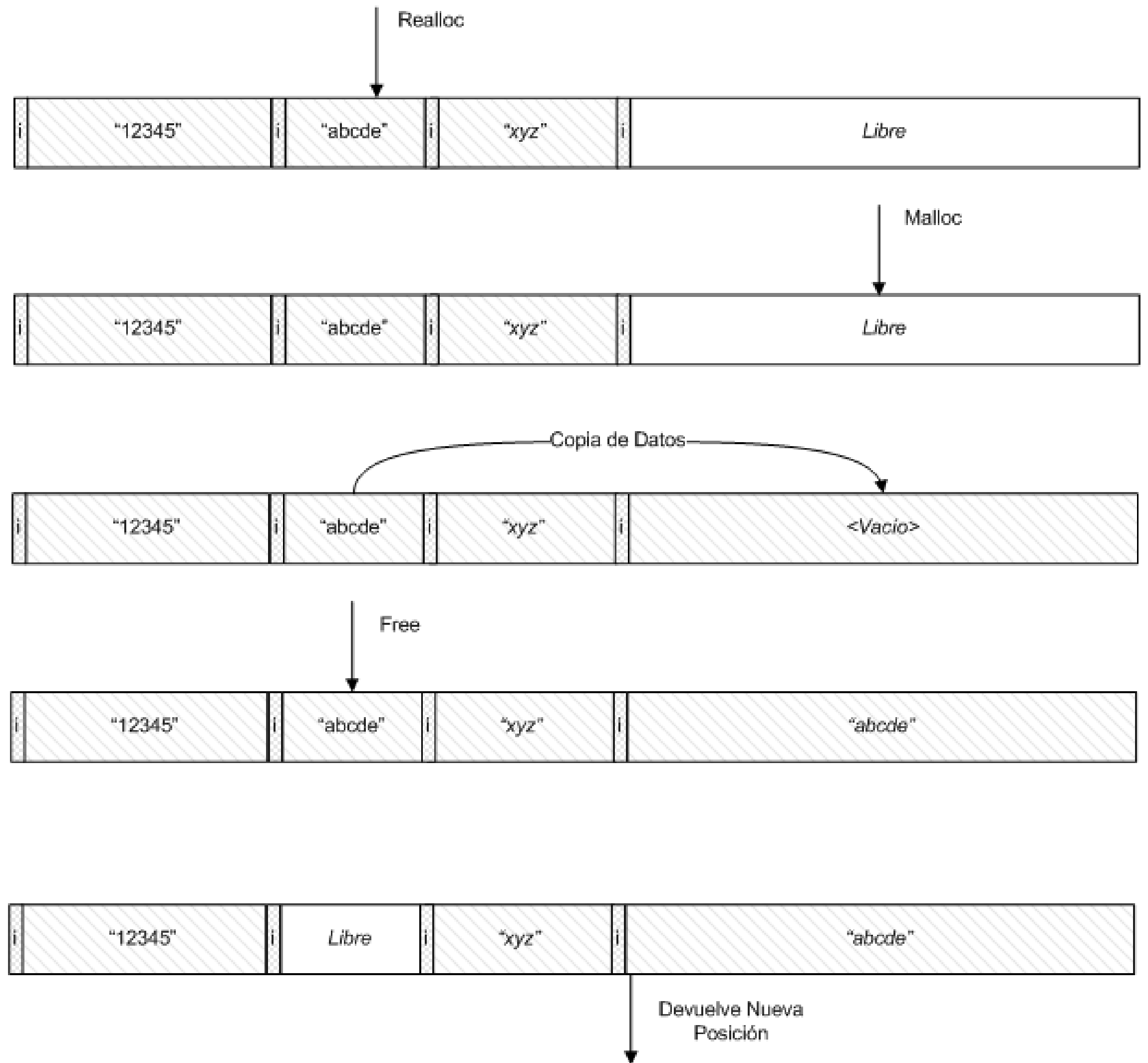

Figura 21: Ejemplo de malloc

En el malloc el primer paso es buscar por un segmento de memoria libre que satisfaga el nuevo tamaño deseado. Esto lo realiza reutilizando la funcionalidad misma del malloc. Cuando malloc le devuelve la nueva dirección, realloc se encarga de copiar los datos presentes en el primer segmento al nuevo. Luego de que se hayan copiado los datos, se libera el primer segmento (se efectua merging si es necesario) y se devuelve la posición del nuevo segmento con los datos del anterior.

- *free(void*):* Su funcionamiento es similar al utilizado por el kernel, se le pasa la dirección inicial del segmento en memoria y se encarga de cambiar su flag de ocupado a 0 y efectuar merging de ser necesario.

- *void iFnListar():* Lista en pantalla la información física del heap usuario y su utilización. Muestra todas las particiones tanto libres como ocupadas (tanto para variable como para fija en sus dos variantes), su tamaño y su estado.

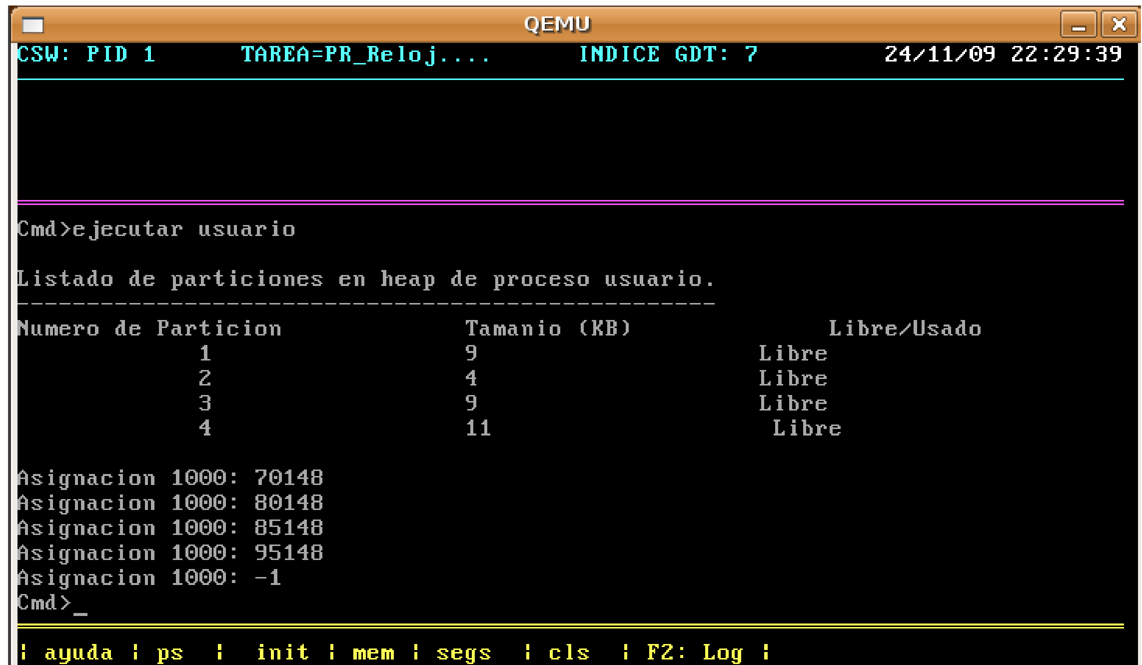

Figura 22: Ejemplo de Lista.

En la figura 22, el proceso usuario lista primero las particiones que tiene tanto disponibles como no disponibles, y su tamaño (hereda las del sistema operativo inicialmente, luego se puede configurar). Luego efectúa mallocs recursivamente hasta recibir un numero -1 que indica que no hay memoria disponible para la asignación.

Se puede ver que solo 4 solicitudes pudieron ser satisfechas debido a que solo contaba con 4 particiones libres, la quinta falla.

- Códigos de error

Los comandos específicados devuelven un código de error según el fallo que haya ocurrido, la siguiente tabla los especifica.

Nivel	Funcion	Código	Causa
Usuario	Malloc	-1	Memoria Insuficiente
Usuario	Malloc	-2	Memoria solicitada > Memoria Fija (en Part. Fija)
Usuario	Realloc	-1	Memoria Insuficiente
Usuario	Realloc	-2	Memoria solicitada > Memoria Fija (en Part. Fija)
Usuario	Realloc	-3	Heap No Inicializado aún.
Usuario	Realloc	-4	Tamaño solicitado < Tamaño Actual (en modo Variable)
Usuario	Free	-1	Partición no encontrada.
Usuario	Free	-3	Heap No Inicializado aún.
Kernel	Malloc	-1	Memoria Insuficiente
Kernel	Malloc	-2	Memoria solicitada > Memoria Fija (en Part. Fija)
Kernel	Realloc	-1	Memoria Insuficiente
	Realloc	-2	Memoria solicitada > Memoria Fija (en Part. Fija)
Kernel	Free	-1	Partición no encontrada.
Kernel	Realloc	-4	Tamaño solicitado < Tamaño Actual (en modo Variable)

Es responsabilidad de la función o proceso que las haya llamado interpretarlos y manejar los errores. Por ejemplo si no se maneja bien una devolución de -1 de un Malloc, e intenta editarse de todas maneras el puntero, esto ocasionará una excepción de chequeo de límites (GF #13) matando al proceso.

- Variables de entorno

Nuestro enfoque busca otorgar al usuario la posibilidad de configurar la mayor cantidad de parámetros en la ejecución y configuración de la administración de memoria del SODIUM. De esta manera quien lo use con objetivos didácticos podrá "meter mano" y afectar el funcionamiento de nuestro algoritmo y otros elementos mas.

La mejor manera que encontramos de lograr esto es reutilizar un concepto ya programado y que en muchos sistemas operativos otorga el mismo propósito: las variables de entorno.

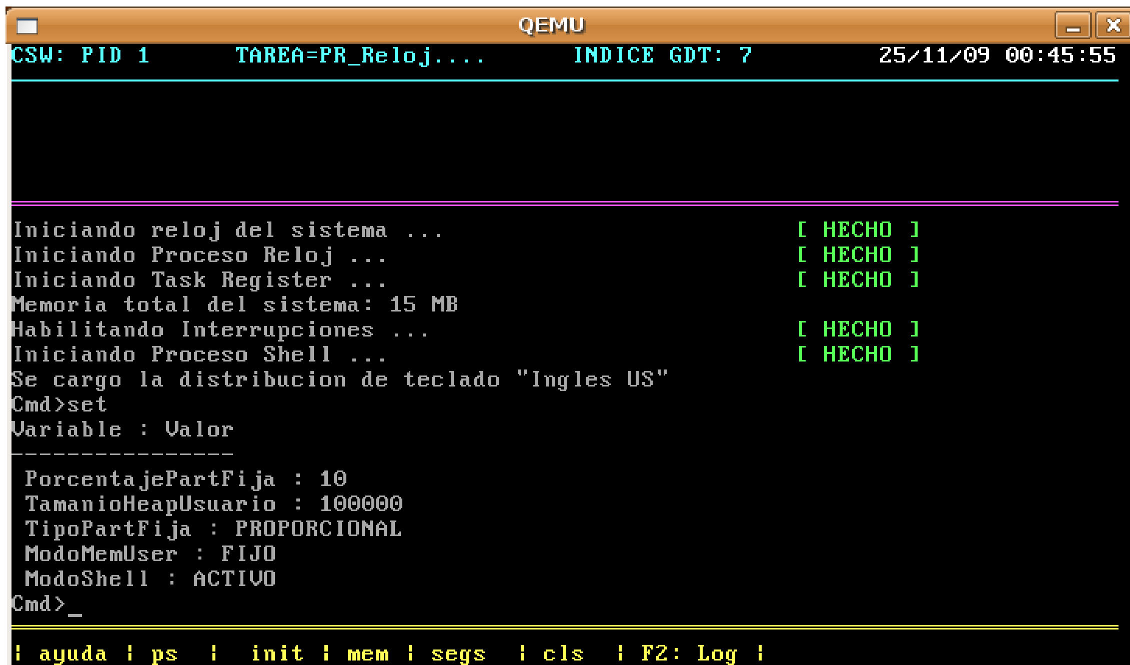

Figura 23: Variables de entorno

Detallaremos mas adelante el propósito y función de cada una de las 5 variables de entorno que utilizamos para configurar nuestra entrega de SODIUM. Pero antes explicaremos la sintaxis para listar, modificar, crear y borrar una variable de entorno.

Para listar variables de entorno:

> set
o alternativamente
> set showall

Para modificar/crear variable de entorno:

> set VARIABLE=VALOR

Por ejemplo para cambiar el valor de ModoShell usaremos la sintaxis:
> set ModoShell=PASIVO

(Nota: Las variables que utilizamos para configuración son case sensitive)

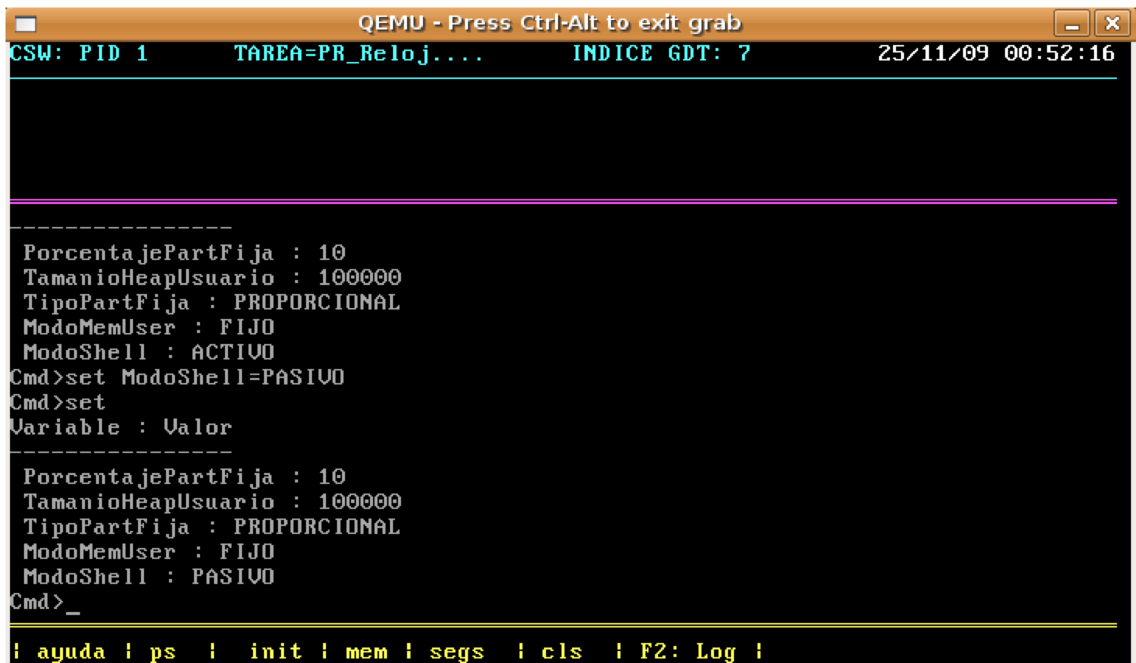

Figura 24: Ejemplo de modificación de variable

Para eliminar una variable de entorno:

> set VARIABLE=eliminate

Por ejemplo para eliminar la variable ModoShell utilizamos la sintaxis:

> set ModoShell=eliminate

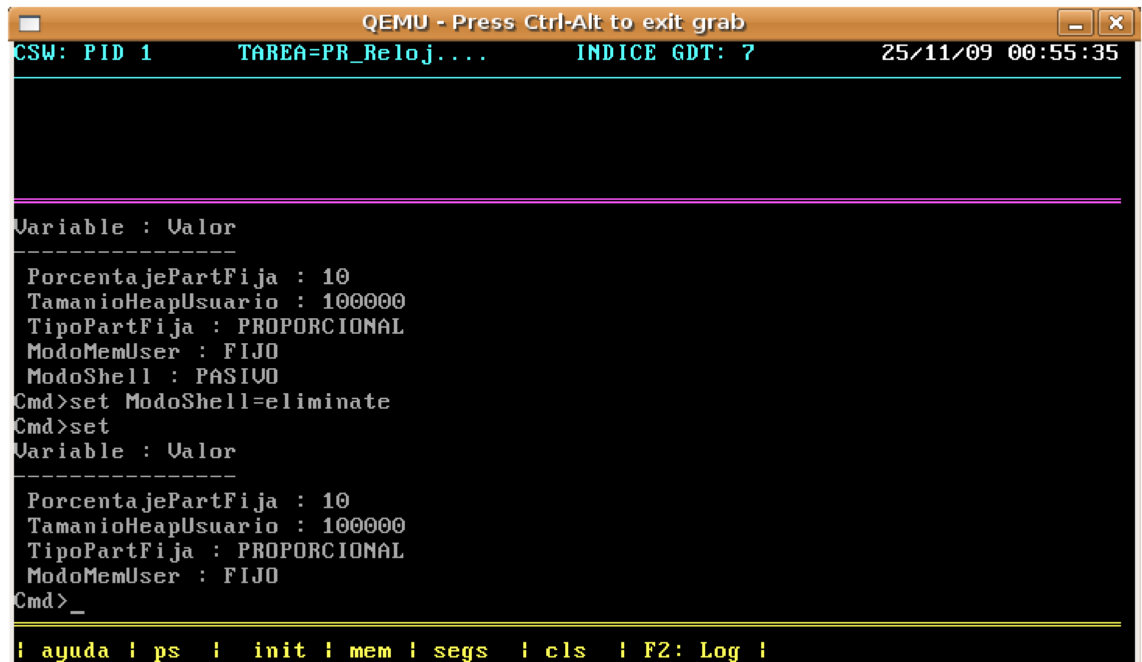

Figura 25: Ejemplo de eliminación de variable

- *Variable ModoMemUser:*

  Indica el modo de memoria en que se va a ejecutar los sucesivos procesos usuario que se creen (hasta que la variable se modifique). Los procesos que ya estan creados no cambian de Modo de ejecución si se cambia esta variable a posteriori.

  Si si valor es “VARIABLE”, entones los sucesivos procesos usuario se ejecutaran utilizando particiones variables. Si es su valor es “FIJO”, los procesos usuario se ejecutaran utilizando particiones fijas.

  Notese que este y los siguientes valores se toman como configuración al proceso al momento de ejecutar el syscall headerinfo() que se ejecuta en InicializarHeapUsuario, tras **el primer malloc efectuado.** Por lo que si se cambió antes que un proceso haya hecho su primer malloc, éste va a iniciar su heap con la nueva configuración.

- *Variable TamanioHeapUsuario:*

  Indica el tamaño del heap de usuario en bytes. De acuerdo con la descripción estructural que ya dimos del proceso usuario en memoria, este tamaño indica la longitud entre el final del stack (comienzo lógico del mismo, final físico) y el límite mismo del proceso. Por lo tanto, esta variable NO equivale al tamaño del proceso, aunque si influye en su tamaño final.

  El tamaño del proceso está definido por la sumatoria entre su tamaño de código, tamaño de área de datos inicializados y no inicializados, un tamaño de stack definido como constante en definiciones.h (recomendamos a la cátedra llevar éste valor a variable de entorno ya que influye mucho en la ejecución de un proceso), y el tamaño del heap definido en nuestro SODIUM como variable de entorno.

- *Variable TipoPartFija:*

  Si se configuró la variable *ModoMemUser=FIJO,* debe indicarse que tipo de partición fija se trata. Si se utiliza *TipoPartFija=FIJO,* el administrador de proceso dividirá al heap en porcentajes de igual tamaño.

  Estos porcentajes pueden configurarse en la variable *PorcentajePartFija,* Por ejemplo, si se define *PorcentajePartFija=10,* el proceso contará con 10 particiones, cada una del tamaño del 10% del tamaño del heap usuario.

  Si se especifica *PorcentajePartFija=15,* se crearán 6 particiones del 15% del tamaño del heap de usuario, y quedará 10% de desperdicio. Invitamos al usuario a hacer la prueba de que esto efectivamente pasa, utilizando la función para listar particiones con las que cuenta la biblioteca de usuario para administración de memoria.

Si se especifica *TipoPartFija=PROPORCIONAL,* el proceso creará sus particiones basado en una lista de porcentajes definida por el usuario. Esta lista se hereda de la configuración del kernel si se eligió setear los porcentajes manualmente, sino puede ser definida luego mediante los siguientes comandos:

> listarporc

El comando listarporc muestra en pantalla la lista de porcentajes de división de heap de proceso. Por defecto hay un solo valor en 100% (un solo segmento usable del tamaño del heap) salvo que se haya configurado manualmente en el menú inicial o se modifique manualmente la tabla luego.

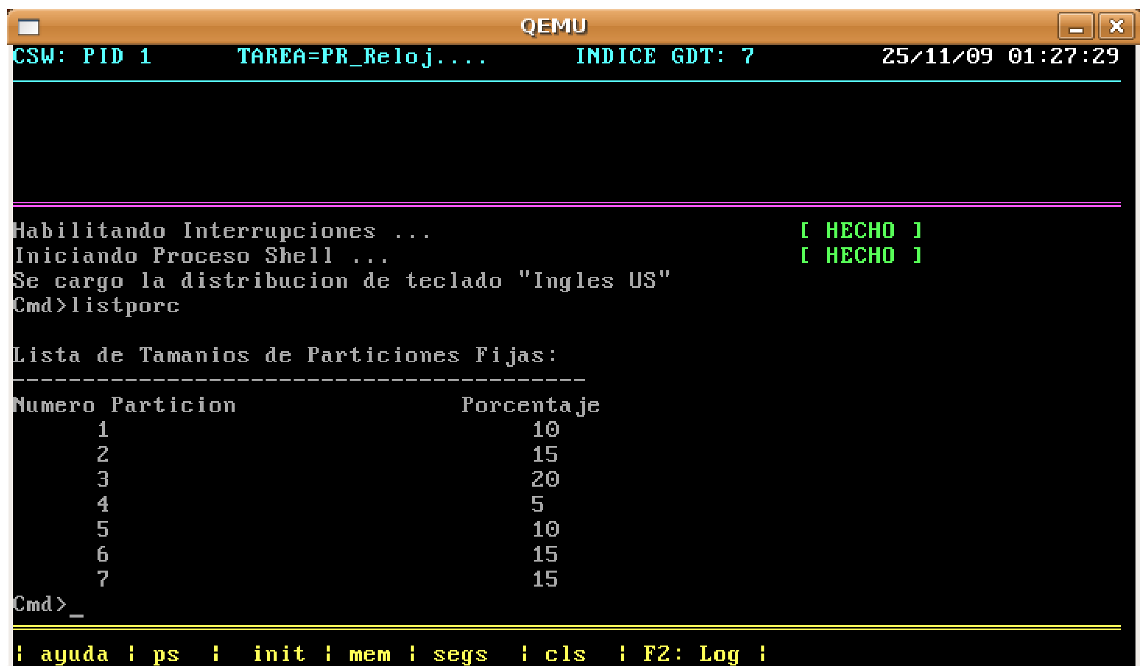

Figura 26: Comando listporc. Lista de porcentajes de particiones.

> borrarpf

El comando borrarpf elimina la última entrada de la lista de porcentajes de particiones fijas proporcionales. Si es la última, reporta que la operación no puede ser efectuada y deja una partición con un porcentaje mínimo para dejar al usuario agregar mas porcentajes.

```
QEMU
CSW: PID 1      TAREA=PR_Reloj....      INDICE GDT: 7       25/11/09 01:30:52

      5                            10
      6                            15
      7                            15
Cmd>borrarpf
Cmd>listporc

Lista de Tamanios de Particiones Fijas:
----------------------------------------
Numero Particion                Porcentaje
      1                            10
      2                            15
      3                            20
      4                            5
      5                            10
      6                            15
Cmd>_
| ayuda | ps  |  init | mem | segs  | cls  | F2: Log |
```

Figura 27: el comando borrapf borró la ultima entrada en la lista de porcentajes

> agregarpf PORCENTAJE

El comando agregarpf permite agregar una nueva partición a la lista de porcentajes con el tamaño en porcentaje pasado por parámetro. Este comando no será exitoso si la lista de porcentajes suma, junto con el pedido mas del 100%. Para poder agregar una nueva partición de tal tamaño debera ejecutarse borrarpf tantas veces como sea necesario.

```
QEMU
CSW: PID 1      TAREA=PR_Reloj....      INDICE GDT: 7       25/11/09 01:35:50

      5                            10
      6                            15
Cmd>agregarpf 10
Cmd>listporc

Lista de Tamanios de Particiones Fijas:
----------------------------------------
Numero Particion                Porcentaje
      1                            10
      2                            10
      3                            15
      4                            20
      5                            5
      6                            10
      7                            15
Cmd>_
| ayuda | ps  |  init | mem | segs  | cls  | F2: Log |
```

Figura 28: el comando agregarpf agrega una nueva entrada al principio de la lista

- *Variable ModoShell:*

  Esta variable indica si luego de hacer un fork / exec, el proceso padre shell debe efectuar un wait_pid sobre el hijo. En una explicación mas sencilla indica si el shell debe esperar a que termine la ejecución de los procesos ejecutados de manera POSIX (ya hemos mencionado que la manera de efectuar este tipo de ejecución es poner el nombre del binario sin el .bin como comando).

  Ejemplo: *>usuario*

  Las dos opciones posibles son:

  ModoShell=ACTIVO
  ModoShell=PASIVO

  En el caso del Shell en modo Pasivo, luego de ejecutarse el fork, el shell efectua un waitpid sobre el proceso nuevo, de manera que se mantendrá en suspenso hasta que el proceso nuevo termine su ejecución.

  En el caso del Shell en modo Activo, se omite la instrucción waitpid y se continua con la recepción de teclado mientras el proceso hijo está en ejecución. Esto puede ser muy útil para obtener información sobre las particiones libres de kernel o cualquier otra cosa mientras se está ejecutando el proceso nuevo.

  Debe tenerse cuidado, sin embargo, que cuando el proceso usuario termine quedará como ZOMBIE (y sus recursos de memoria aún asignados) a menos que desde el shell se use el comando syswaitpid permitiendole al shell recuperar la señal SIGCHLD y al kernel limpiar el proceso de memoria.

- Syscalls BRK y SBRK

Para la definición de estos syscalls tomo prestado la definición en el apunte de memoria la cátedra.

> "*Un proceso puede modificar el tamaño de datos, para ello se utiliza la llamada al sistema brk, cuya sintaxis es la siguiente int brk(void *end_data_segment), donde el parámetro end_data_segment especifica la dirección final del segmento de datos, debe ser superior a la dirección del segmento de código e inferior en 16 Kbytes a la dirección del fin de segmento de pila. La otra llamada que le permite modificar el tamaño del segmento de datos es sbrk y la sintaxis correspondiente es void *sbrk (ptrdiff_t increment) el parámetro increment indica el número de bytes a añadir o quitar del segmento, si es positivo o negativo respectivamente. Ambas cuando fallan devuelven 1.*"

Para ilustrarlo, podemos ver en la siguiente figura una representación del estado de un proceso cuando efectúa un malloc pero no hay espacio suficiente para la asignación. Vale destacar que para efectuar un brk o sbrk no es necesario que el heap esté lleno, sino que sucede cuando no hay espacio para satisfacer la petición del malloc.

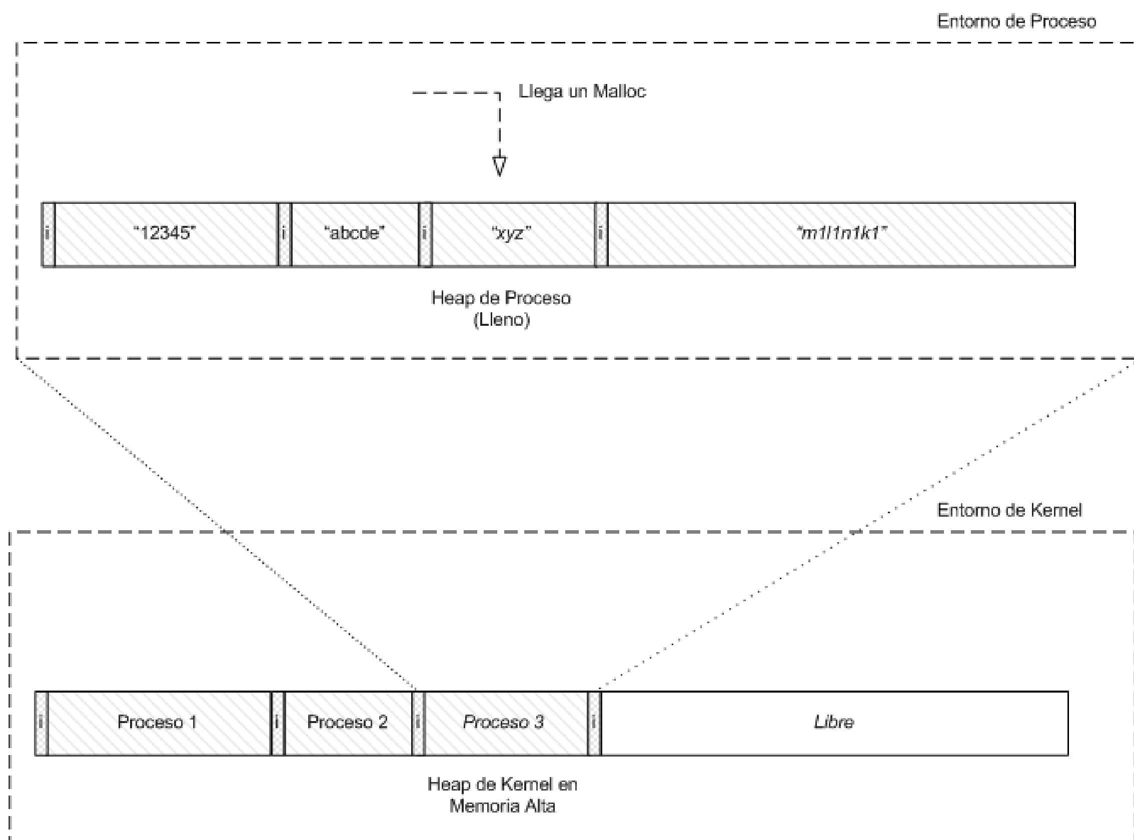

Figura 29: Situación en la que un proceso debe utilizar el syscall sbrk

A nivel de heap kernel, la petición de sbrk o brk se maneja como un realloc del segmento asignado al proceso hacia un nuevo segmento que pueda satisfacer la necesidad del incremento de tamaño.

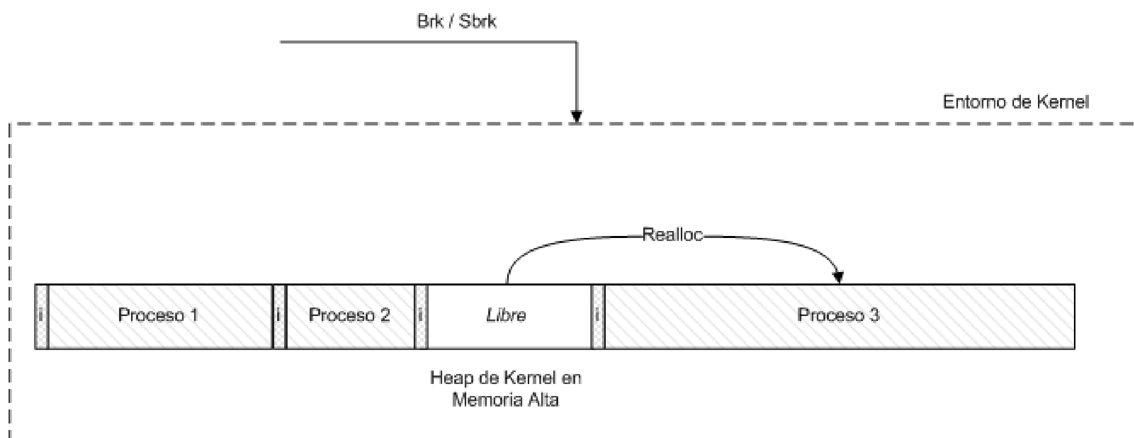

Figura 30: Realloc a nivel kernel

A nivel de administración de proceso, se actualiza la entrada del PCB con la nueva información de comienzo de segmento y la base y limites de la entrada GDT del proceso para que cuando se le devuelva el control, el cpu le permita ejecutar en el nuevo entorno.

En la interfaz de usuario del algoritmo, el malloc intenta reservar memoria una vez, si falla una vez, intenta efectuar un sbrk. Si este devuelve un mensaje de error, se devuelve un -1 al proceso indicando que no hay memoria suficiente para la reserva.

Si el sbrk fue exitoso, devuelve el nuevo tamaño del heap (tamaño anterior + tamaño petición de malloc). Este es guardado por el administrador como el nuevo tamaño del heap propio, también asigna al último segmento nuevo como libre. y reintenta un malloc (recursivo) en el nuevo segmento.

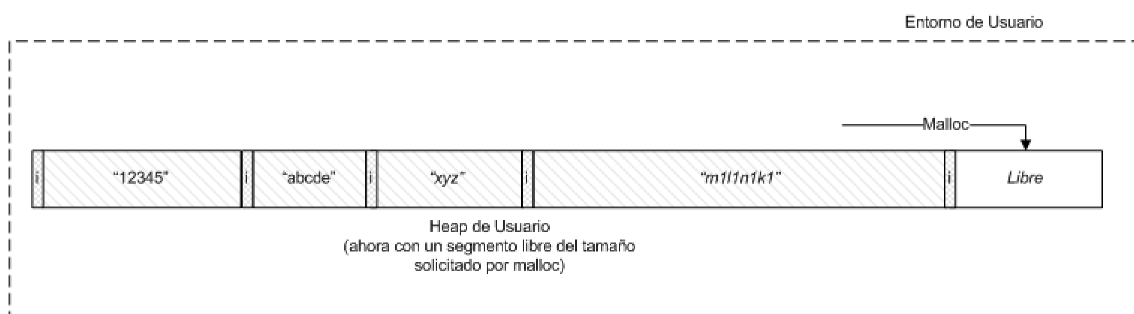

Figura 31: El proceso intenta el malloc luego del sbrk

Si bien los syscalls sbrk/brk se pueden implementar tanto en partición variable como en particiones fijas equitativas (y de hecho está implementado para los dos) hemos decidido comentar el código de sbrk de las particiones fijas porque no respondería a cómo los sistemas operativos viejos lo implementaban y podría confundir el concepto a quienes lo ejecuten. Si el proceso no maneja particiones variables, no se efectua sbrk y se devuelve un -1.

FORMATO ELF

- Introducción

El formato de archivo objeto ELF (Executable and Linkable Format), diseñado por Unix System Laboratories, es actualmente el formato más utilizado en Linux. A diferencia de otros formatos de archivo propietarios, ELF es muy flexible y no está atado a ninguna rquitectura de procesador en particular. Esto ha permitido que sea adoptado rápidamente por muchos sistemas operativos y plataformas diferentes.

El formato ELF ha reemplazado a formatos ejecutables más antiguos, como "a.out" y "COFF". ELF ayuda a los desarrolladores al proveer un conjunto de definiciones de interfaces que son idependientes de la plataforma. Esto hace que se reduzca la necesidad de modificar código y recompilar los programas. Algunos ejemplos de herramientas que utilizan el formato ELF son los compiladores, debuggers y linkers entre otros.

- Formato ELF

La especificación ELF, describe tres tipos de archive objeto.

- Archivos reubicables: Contienen código y datos, y pueden ser linkeados con otros archivos objeto. Estos archivos también son conocidos como archivos objeto (archivos con extensión .o).
- Archivos ejecutables: Contienen código y datos que pueden ser ejecutados.
- Archivos objeto compartidos: Contienen información reubicable que puede ser compartida con otros objetos compartidos, de manera estática o dinámica.

Un objeto ELF consiste de una cabecera ELF obligatoria que lo identifica como tal, seguida de contenido opcional como una Program Header Table y una Section Header Table:

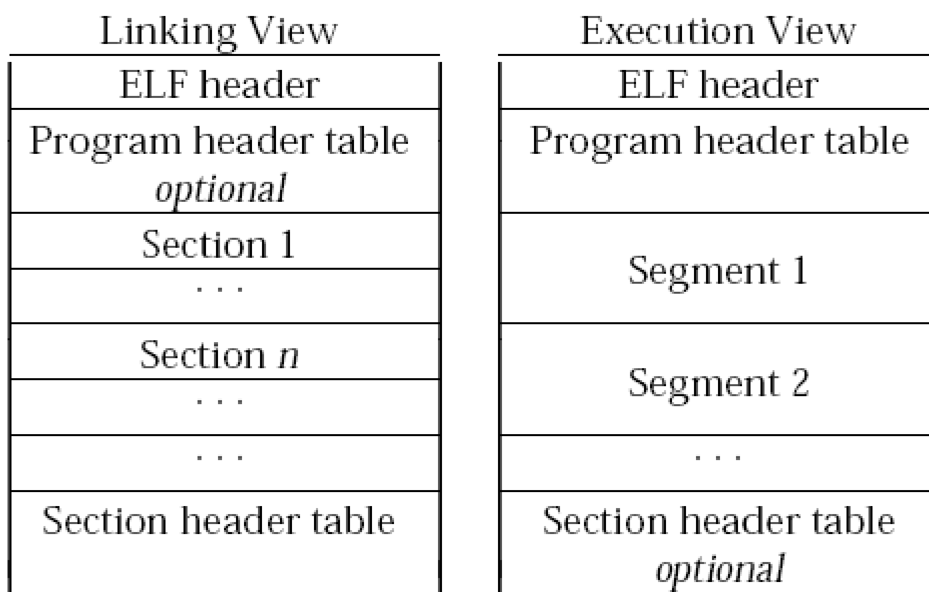

Figura 32: Formatos del ELF según etapa (linkeo y ejecución)

La cabecera ELF define la estructura del resto del archivo. Esta cabecera está siempre presente en un archivo ELF válido. Describe la clase de archivo (64 o 32 bits), el tipo (reubicable, ejecutable u objeto compartido), y el orden de los bytes utilizado (little endian o big endian).
La Program Header Table está presente en objetos ejecutables y contiene información necesaria para el programa al momento de la carga.
Los contenidos de un objeto ELF reubicables están contenidos en las secciones ELF. Estas secciones aparecen como entradas en la Section Header Table, que contiene una entrada por sección presente en el archivo.

- Cabecera ELF

A continuación describimos la estructura de la Cabecera ELF.

```
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf_Ehdr;
```

- Los bytes iniciales son los que marcan al archive como un archive objeto. Estos bytes contienen información independiente de la máquina, con la cual se interpreta el contenido del archivo.
- E_type: identifica el tipo de archivo, como se muestra en la siguiente tabla:

Nombre	**Valor**	**Significado**
ET_NONE	0	Ningún tipo de archivo
ET_REL	1	Archivo reubicable
ET_EXEC	2	Archvio ejecutable
ET_DYN	3	Archivo objeto compartido
ET_CORE	4	Archivo core
ET_LOPROC	0xff00	Específico del Procesador

Nombre	Valor	Significado
ET_HIPROC	0xffff	Específico del Procesador

Aunque el contenido de los Archivos cores no está especificado, el tipo ET_CORE se reserva para marcar a este tipo de archivo. Los valores desde ET_LOPROC hasta ET_HIPROC inclusive se reservan para semánticas específicas del procesador. Otros valores se reservan para otros usos.

- E_machine: especifica la arquitectura requerida por el archivo. En la siguiente tabla se listan las arquitecturas más relevantes:

Nombre	Valor	Significado
EM_NONE	0	Ninguna máquina
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_SPARC32PLUS	18	Sun SPARC 32+
EM_SPARCV9	43	SPARC V9
EM_AMD64	62	AMD 64

- E_version: identifica la version del archive, según los valores de la siguiente tabla:

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	>=1	Current version

- E_entry: Dirección virtual a la cual el sistema le transfiere el control primero, iniciando el proceso. Si el archivo no tiene un punto de entrada asociado, este campo tiene valor 0.
- E_phoff: Muestra el Offset en bytes de la Program Header Table. Si el archivo no tiene Program Header Table, tiene valor 0.
- E_shoff: Muestra el Offset en bytes de la Section Header Table. Si el archivo no tiene Section Header Table, tiene valor 0.
- E_flags: flags específicas del procesador asociadas con el archivo. Este campo es cero para x86.
- E_ehsize: tamaño de la cabecera ELF en bytes.

- E_phentsize: tamaño en bytes de cada entrada de la Program Header Table. Todas las entradas tienen el mismo tamaño.
- E_phnum: cantidad de entradas en la Program Header Table.
- E_shentsize: tamaños de una cabecera de sección en bytes. Una cabecera de sección es una entrada en la Section Header Table. Todas las entradas tienen el mismo tamaño
- E_shnum: cantidad de entradas en la Section Header Table.
- E_shstrndx: el índice de la entrada de la Section Header Table, asociada con la tabla de strings con los nombres de las secciones.

Program Header Table

Cuando se carga un ejecutable en memoria, el sistema operativo lo ve como un conjunto de "segmentos". Cada uno de esos segmentos comienza en algún lugar del ELF, tiene asociada una protección particular (solo lectura, lectura-escritura), y se carga en una dirección específica de memoria.
La Program Header Table contiene información sobre los segmentos presentes en el archivo ELF. Usando la Program Header Table, el archivo ELF puede ser visto como un conjunto de segmentos que no se solapan. Cada uno de esos segmentos se describe en una entrada de la Program Header Table.
A continuación se describe la estructura de una entrada de la Program Header Table:

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

- P_type: mantiene el tipo de entrada de la Program Header Table. Por ejemplo:
    - Un segmento de tipo LOAD se carga en memoria.
    - Un segmento de tipo NOTE contiene información auxiliar.
    - Un segmento de tipo PHDR describe a la propia Program Header Table.

La especificación ELF mantiene los valores desde 0x60000000 a 0x6FFFFFFF para información privada del sistema operativo y los valores 0x70000000 hasta 0x7FFFFFFF para información específica del procesador.

- P_offset: contiene el Offset en el objeto ELF donde comienza el segmento descripto por la entrada de la tabla.
- P_vaddr: La dirección virtual a la que el segmento debería ser cargado.
- P_paddr: La dirección física a la que el segmento debería ser cargado. Este campo no se utiliza para objetos de usuario.
- P_filesz: cantidad de bytes que el segmento ocupa en el archivo. Este valor es 0 para segmentos que no tienen datos asociados en el archivo.
- P_memz: cantidad de bytes que el segmento ocupa en memoria.
- P_flags: flags adicionales que especifican propiedades del segmento.
- P_align: alineamiento requerido por el segmento tanto en memoria como en el archivo. El valor de este campo es un potencia de 2.

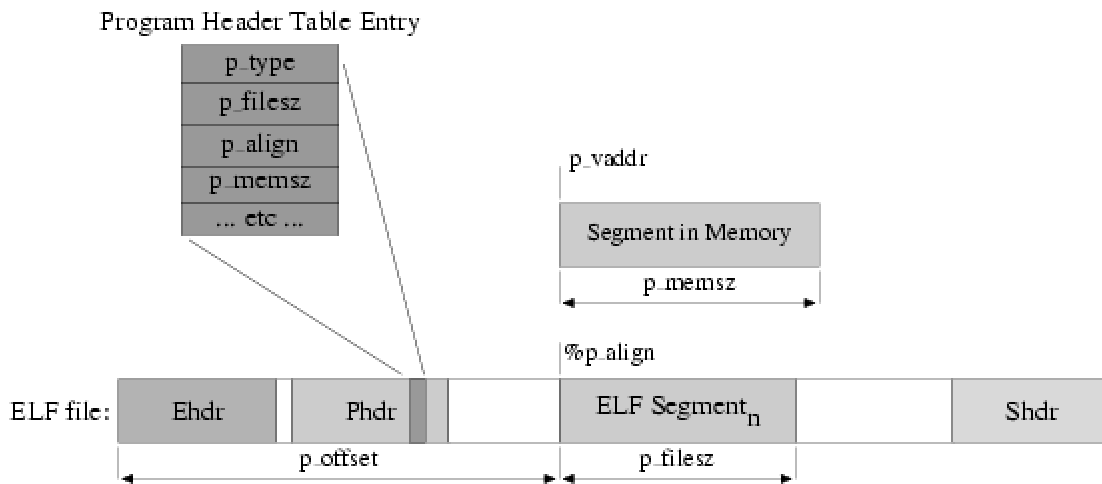

Figura 33: Contenido de la cabecera ELF

Section Header Table

Para la linkedición, los objetos ELF están agrupados en secciones. Cada sección del ELF representa un tipo de información. Por ejemplo una sección puede contener una tabla de strings usada para los símbolos del programa, otra puede contener información de debug, y otra puede contener código máquina. Las secciones no vacías no se solapan en el archivo.
Cada sección es descripta por una entrada en la Section Header Table. Esta tabla usualmente se ubica al final del objeto ELF.
A continuación se describen los elementos de una entrada de la Section Header Table:

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

- Sh_name: se usa para codificar el nombre de la sección. Como los nombres de sección son strings de tamaño variable, no se mantienen en la Section Header Table, si no que se mantienen en la tabla de strings que mantiene los nombres de las secciones, y el campo sh_name de cada cabecera de sección contiene un índice de la tabla de strings. La cabecera ELF contiene en el campo e_shstrndx el índice de la tabla de strings dentro de la Section Header Table.
- Sh_type: Especifica el tipo de la sección. Por ejemplo una sección de tipo SHT_PROGBITS contiene código ejecutable, mientras que una sección de tipo SHT_SYMTAB es una sección que contiene una tabla de símbolos. La especificación ELF reserva los valores entre 0x60000000 y 0x6FFFFFFF para especificar tipos de sección específicos del sistema operativo, y los valores entre 0x70000000 y 0x7FFFFFFF para tipos de sección específicos del procesador. Además los valores entre 0x80000000 y 0xFFFFFFFF se han asignado a las aplicaciones para su propio uso.

- Sh_flags: indican si una sección tiene propiedades específicas. Los valores entre 0x00100000 y 0x08000000 son utilizados para uso específico del sistema operativo. Los valores entre 0x10000000 y 0x80000000 son reservados para uso específico del procesador.
- Sh_addr: si la sección va estar en la imagen en memoria de un proceso, especifica la dirección en la que debería estar el primer byte. De lo contrario tiene el valor 0.
- Sh_offset: Este valor especifica el Offset en bytes desde el comienzo del archivo hasta el comienzo de la sección.
- Sh_size: especifica el tamaño de la sección en bytes.
- Sh_link y sh_info: contienen información adicional específica de la sección.
- Sh_addralign: para secciones que tienen requerimientos de alineamiento específicos. Su valor es una potencia de 2.
- Sh_entsize: para secciones que contienen arrays de elementos de tamaño fijo, este campo especifica el tamaño de cada elemento.

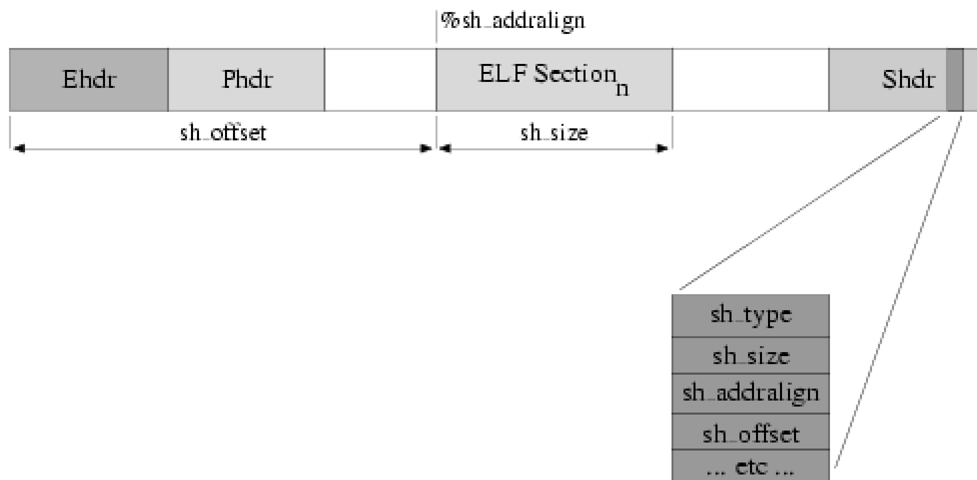

Figura 34: Contenido de la cabecera ELF

<u>- ELF con swap</u>

Cuando el sistema crea o aumenta una imagen de proceso, lógicamente copia los segmentos del archivo en segmentos de memoria virtual. Si se lee físicamente el archivo o no, y en qué momento, depende de la ejecución del programa, su carga, etc. El proceso no requiere una página física a menos que se referencie la página lógica durante la ejecución, y los procesos usualmente dejan varias páginas sin referenciar.
En la práctica, los archivos ejecutables y objetos compartidos deben tener imágenes cuyos offsets y direcciones virtuales sean congruentes módulo tamaño de página.
A continuación vemos un ejemplo para explicar mejor el tema.

File Offset	File	Virtual Address
0	ELF header	
Program header table		
	Other information	
0x100	Text segment . . . 0x2be00 bytes	0x8048100 0x8073eff
0x2bf00	Data segment . . . 0x4e00 bytes	0x8074f00 0x8079cff
0x30d00	Other information . . .	

Figura 35: Swap con ayuda de virtual address para swap

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x1000	0x1000

Figura 36: Correlación entre miembros de la información de datos y código

Aunque el archivo del ejemplo tiene offsets y direcciones virtuales congruentes módulo 4 KB para data y text, hasta 4 páginas del archivo tienen text o data impuros (dependiendo del tamaño de la página y bloque del file system). La primera página de text contiene la cabecera ELF, la Program Header Table y otra información. La última página de text contiene una copia del comienzo de los datos. La primera página de data tiene una copia del final del text. Y la última página de data contiene información adicional irrelevante para correr el proceso.
Lógicamente, el sistema establece los permisos de memoria como si cada segmento fuera completamente separado. Las direcciones de los segmentos se ajustan para asegurar que cada página lógica en el espacio de direcciones tiene su propio conjunto de permisos. En el ejemplo, la región del archivo que contiene el final del text y el comienzo del data, será mapeado dos veces, a una dirección virtual para el text y a otra para el data.

- ELF en Sodium

En esta entrega del sodium logramos compilar satisfactoriamente un proceso (todos los procesos usuario) con cabecera ELF, cargar esta cabecera, obtener datos útiles de esta y hacer funcionar el proceso normalmente. Esto representa una optimización importante respecto al anterior método de determinar la información de tamaño de proceso.

A continación se detalla la forma para lograr una correcta compilación del formato ELF en Sodium

ELF es el formato en el que gcc compila por defecto. En el proceso de compilación del SODIUM, luego de realizar la compilación y la linkedición, se ejecuta el comando objcopy, mediante el cual se le quitan algunas secciones innecesarias del ELF, y se le quitan las cabeceras y estructuras propias del ELF y se los transforma en un binario puro.
Para poder compilar los archivos .bin en formato ELF entonces, solo basta con quitar del comando objcopy la opción "-O binary". De esta manera mantenemos la estructura del ELF (cabecera, Program Header Table, etc.).
Luego será necesario adaptar el SODIUM para cargar un archivo ELF de la manera que se explica más adelante.

Para cargar un proceso utilizando la información del formato ELF utilizamos las siguientes estructuras definidas en pcb.h

```
typedef struct {
        unsigned char       ident[16];
        unsigned short      type;
        unsigned short      machine;
        unsigned int version;
        unsigned int entry;
        unsigned int phoff;
        unsigned int sphoff;
        unsigned int flags;
        unsigned short      ehsize;
        unsigned short      phentsize;
        unsigned short      phnum;
        unsigned short      shentsize;
        unsigned short      shnum;
        unsigned short      shstrndx;
} elfHeader;

typedef struct {
        unsigned int name;
        unsigned int type;
        unsigned int flags;
        unsigned int addr;
        unsigned int offset;
        unsigned int size;
        unsigned int link;
        unsigned int info;
        unsigned int addralign;
        unsigned int entsize;
}SHeader;
```

```
typedef struct {
        unsigned int type;          //Elf32_Word p_type;
        unsigned int offset;        //Elf32_Off  p_offset;
        unsigned int vaddr;         //Elf32_Addr p_vaddr;
        unsigned int paddr;         //Elf32_Addr p_paddr;
        unsigned int filesz;        //Elf32_Word p_filesz;
        unsigned int memsz;         //Elf32_Word p_memsz;
        unsigned int flags;         //Elf32_Word p_flags;
        unsigned int align;         //Elf32_Word p_align;
} PHeader;


/**
 \brief Estructura que recibe la cabecera ELF de un proceso

*/
typedef struct {
   elfHeader elfHeaderData;
   SHeader   SHeaderData;
   PHeader   PHeaderData;
   unsigned char ucbuffer[4];
} stuHeaderELF;
```

En la función iFnLeerCabeceraEjecutable en gdt.c leemos primero la parte del binario que contiene la cabecera ELF:

```
/* Estoy en el punto inicial frente a la cabecera ELF. La voy a leer. */

     ucpFnCopiarMemoria((unsigned char*)ucLecturaTemporalELF,(unsigned char*)pstuInfo->pvPuntoCarga, TAM_ELF);
```

De esta estructura se obtienen los datos de tamaño de código exactos del proceso:

```
stELF = &ucLecturaTemporalELF;

pstuInfo->uiTamanioTexto  = stELF->SHeaderData.size;
pstuInfo->uiTamanioDatosInicializados = 0;
pstuInfo->uiTamanioStack  = STACK_SIZE;
```

Entonces para crear el segmento en memoria, se utilizan los datos obtenidos en el ELF mas el tamaño determinado por el usuario para el tamaño de heap:

```
   uiTamSegCodigo     =
REDONDEAR_HACIA_ARRIBA_A_4K(stuInfoExe.uiTamanioTexto);
   uiTamStack         =
REDONDEAR_HACIA_ARRIBA_A_4K(stuInfoExe.uiTamanioStack);
   uiTamInicializados = stuInfoExe.uiTamanioDatosInicializados;
   uiTamHeap          = iFnCtoi(cpFnGetEnv("TamanioHeapUsuario"));
   uiTamSegDatos      =
REDONDEAR_HACIA_ARRIBA_A_4K(uiTamSegCodigo + uiTamStack +
uiTamInicializados + uiTamHeap); //Se crea CON heap
```

PARTE FINAL

Conclusión

Desde la versión pasada por la cátedra a principios de año, hasta nuestra versión de esta entrega, el SODIUM ha evolucionado en casi todos los aspectos posibles. En el trabajo práctico N°1 logramos mejorar la seguridad del sistema operativo separando los ámbitos de Kernel y Usuario (detalles sobre todas las optimizaciones efectuadas en el mismo están en su documentación)

En esta versión se han refinado muchas funcionalidades y configuraciones del Sodium para hacerlo mucho mas estable. Funcionan todas las funcionalidades en todos los emuladores (Qemu, Bochs, VirtualBox, y VMware) y ha sido probado con éxito en computadoras físicas tanto de 32 como 64 bits, single y dual-core y con tamaños de 16, 32 y 64 megas de memoria.

Nuestro objetivo desde el comienzo de este trabajo práctico fue dejar al SODIUM en las mejores condiciones posibles de funcionamiento, para que el alumnado de años siguientes no debe atravesar dificultades de base (como tuvimos que hacer nosotros) para la implementación de nuevas funcionalidades.

Ahora el sodium cuenta con un administrador nuevo de memoria particionada que asegura una utilización eficiente del heap de kernel tanto como de usuario, y cuenta con una base estable para nuevos desarrollos.

Bibliografía

- Apuntes de la cátedra.
- Documentación automática provista por el Doxygen
- Stallings W. "Sistemas Operativos" Edición 2da

Anexo: Historial de Modificaciones.

- Devolver el sodium al manejo de procesos con un solo segmento de memoria para cada uno (a este punto estamos usando 2 segmentos, uno para codigo y otro para datos) ya que el trabajo practico, y la càtedra exigen el uso estricto de Particion fija / Particion variable. [HECHO]

Existen un par de problemas que surgiràn de esto:

* Algunas funciones de shell estan implementadas para usar un DS. Aunque en teorìa esto no tendrìa ningun problema, porque a pesar de que ahora compartiran el mismo segmento de memoria, la relaciòn DS+Offset de una variable no se va a ver afectada ya que Comienzo de CS = Comienzo de DS.

Queda para investigar para la catedra si se desea crear segmentaciòn la compilación inteligente de manera de usar CS para constantes y DS para variables. [HECHO]

De todas formas hay que adaptar gran parte del shellusuario para que trabaja con un solo segmento. [HECHO]

* El pasaje de argumentos en C se hace en las primeras posiciones del DS. Al compartir el mismo comienzo el DS y el CS (esto podrìa no ser así... será algo que investigaremos mas adelante) el hecho de pasar un parametro estaría pisando una parte del codigo
* Eliminar el syscall char2int auxiliar para obtener el int de un char cuando el char era variable en DS [HECHO]
* Posibles errores imprevisibles que iremos comentando.

- Eliminar la molesta cantidad de binarios compilados que entran en el diskette ya que exeden el espacio de memoria (todo se carga a memoria) destinado a partes escenciales del Sodium como el stack para intercambio.

Esto hay que hacerlo con cuidado de no "maltratar" los Makefiles, ni dejar basura. [HECHO]

- Eliminar los callgates que fueron necesarios para hacer el navegador de niveles. Una vez demostrado que puede navegarse a cualquiera de los 4 niveles de privilegio, solo quedan segmentos de còdigo en memoria que ocupan espacio y entradas en le GDT. Todavía puede mantenerse el navegador en 3-0-3 como demostración de la funcionalidad de los callgates, pero es necesario remover la mayor cantidad de "basura" residual de pruebas y demostraciones. [HECHO]

- Mover los .h de Usuario a la carpeta include/usr [HECHO]

- Efectuar correcciones del TP1 hechas por la catedra
  * Arreglar el problema del buffer de teclado que quedaba como el ultimo enter "presionado" [HECHO]

- Numerar los syscalls nuevos que no son posix en slots no ocupados por syscalls posix. [HECHO]

- Realizar Autocompletar [HECHO]

- Atrapar tecla F2 [HECHO]

- Hacer que el shell utilice espera pasiva. [HECHO]

- Arreglar Fork y Exec [HECHO]

- Los requerimientos de memoria del disco ram es mayor al tamaño de Heap de Kernel (esto podrìa causar que choque contra el stack de kernel en un futuro muy cercano). Si se analiza esta es una pésima decisión ya que el tamaño de un disco por mas minimo que sea excede los 640KB, la proxima particion se reserva después de esto así que falla seguro.
  Lo pasamos a Heap de usuario que tiene mucho mas espacio.

- La memoria es solo usable hasta 31mb.

- Listo el menu para seleccion de tamanios de Particiones de Kernel en memoria alta. En memoria baja se elije el tamaño, no tiene sentido crear tamanios diferentes acà.

- Agregado comando listpart para listar particiones de kernel de memoria alta en uso

- Agregado comando listporc para listar porcentajes de particiones fijas para usuario.

- Agregado comando borrarpf para borrar el ultimo porcentaje de particion fijas para usuario.

- Particion fija variable completado en kernel y en usuario. :D

- Mejorado esteticamente el menu de part fija

- Falta preguntar por brk en procesos con part fija.

- FORK funcionando.

- EXEC funcionando.

- La lista de espera está a medio implementar, hace falta un buen caso de prueba y el chequeo post free.

- Solucionado un problema de base del sodium: El tamaño de stack de usuario era muy chico!, las sucesivas funciones se comian el codigo. Aumentado el stack de 4k a 16k. Ahora debe ser mas que suficiente si se programa bien

- Los procesos que efectuen fork o exec se ponen en espera si no hay memoria para asignarle

- El shell tiene dos modos de funcionamiento determinado por la variable de entorno:

  * ModoShell=ACTIVO: El shell sigue activo despuès de ejecutar un proceso.
    Requiere el uso del comando syswaitpid #NUM_PID para evitar que el proceso siga como
    zombie.
  * ModoShell=PASIVO: El shell espera a que el hijo termine antes de seguir recibiendo de teclado.

- Para ejecutar un proceso en modo POSIX (Fork -> Exec -> Waitpid) solo basta poner el nombre del ejecutable
  sin .BIN.

  ejemplo: $usuario

- Para ejecutar un proceso directamente usar $ejecutar usuario

- Listo, ahora si fork o exec solicitan un segmento y no hay, se pone en espera el proceso. Cuando se libera algun segmento, se les asigna.
  Utilizan espera pasiva. Su estado es de "suspendido" y efectuan un while(1) intentando la operacion y si esta falla, se hace un yield llamando
  al planificador. De esta manera se intenta una vez por vez nada mas.

- Arreglado el nombre de brk a sbrk y creado el syscall brk correspondiente

- Nombrado como firstfit a nuestro algoritmo de memoria

- Impleamentada cola de espera. el comando listaesp sirve para ver que procesos esperan por entrar.
  Los nuevos procesos son mas prioritarios que la espera por fork y exec

- Agregados los comandos a ayuda

- Ahora FIJO_FIJO se llama FIJO_EQUITATIVO y FIJO_VARIABLE se llama FIJO_PROPORCIONAL