Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Sistemas Operativos 1
Catedrático: Ing. Rene Ornelis
Tutor académico: Brian Matus

SECCION A

Vacaciones Segundo Semestre

2024

MANUAL TÉCNICO
Proyecto 2

Nombre
Moises David Maldonado de León

Carné
202010833

Guatemala, 26 de diciembre de 2024

Introducción

En el presente documento, se describe la lógica, estructura y los demás aspectos técnicos del proyecto 2 del curso de Sistemas Operativos 2, el cual se basa en la expansión del Kernel de Linux con nuevas funcionalidades.

A través de la documentación oficial y el desarrollo de habilidades de interpretación y correcta comprensión de los archivos que conforman el Kernel se implementa una nueva llamada al sistema que es accesible desde el espacio de usuario. Por otro lado, se implementan tres módulos cargables del Kernel para la obtención de información con el fin de poder analizar y monitorear distintas métricas de memoria, uso de cpu o y manejo de asignación de memoria.

OBJETIVOS

Generales

Familiarizar al lector con la lógica planteada para el funcionamiento del sistema realizado mediante el uso del lenguaje de programación C y python así como la documentación del Kernel y la configuración de archivos y modificaciones hechas para la creación de nuevas funcionalidades utilizados y módulos cargables del sistema.

Específicos

- Mostrar la estructura de las llamadas realizadas.
- Dar a conocer con detalle las modificaciones y nuevos archivos que se crearon para este proyecto.
- Mostrar los distintos módulos implementados para el sistema.

Alcances

El proyecto se realizó con el fin de comprender el manejo eficiente para la creación de nuevas llamadas al sistema, así como de módulos cargables. Con esto se comprende los pasos para tener control total sobre un sistema Linux y su Kernel, para manejar de manera cuidadosa y metódica, entendiendo la importancia de cada cambio y su impacto. Se sigue expandiendo mediante una función que asigna memoria y unos módulos que permiten el monitoreo de memoria y uso de CPU.

Lógica de programación

Configuración del entorno

Se tiene la siguiente configuración de archivos:

- Modulos: Esta carpeta contiene los módulos utilizados para invocar a la nueva llamada del sistema así como verificar las estadísticas.

- Archivos: Aquí se encuentran los archivos modificados del Kernel de Linux.

- pythonStats: Programa hecho en Python para mostrar las estadísticas de forma estilizada.

- Testing: Carpeta con programas de prueba que no se usarán para la prueba final.

- miScript.sh: Script en bash para simular el funcionamiento “watch” y monitorear las estadísticas.

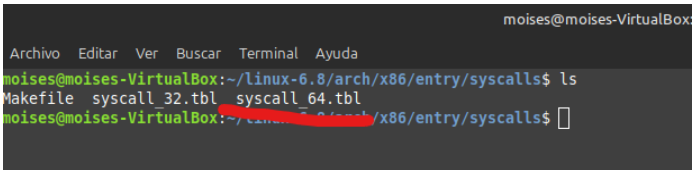
Modificaciones en el Kernel - Nueva llamada al sistema

Tamalloc:

Este algoritmo es una variación del asignador de memoria malloc. Su principal diferencia es la inicialización del espacio de memoria en 0. En esto tiene más parecido a kzalloc, calloc y parecidos. La diferencia principal con estos asignadores de memoria, es qué Tamalloc debe estar diseñado para NO reservar páginas físicas inmediatamente al momento de asignar memoria. Debe ser hasta el primer acceso a cada página (ya sea escritura o lectura) que cause un page fault, que esta región de página debe ser inicializada en 0.

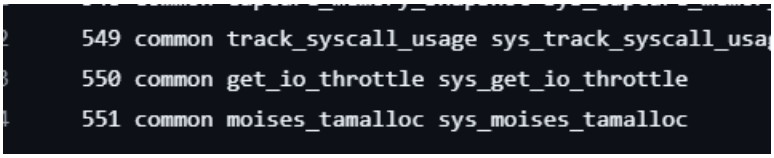
Flujo:

0. **Escoger un nombre moises_tamalloc**
1. **Vamos a la siguiente ruta y a editar el siguiente archivo:**



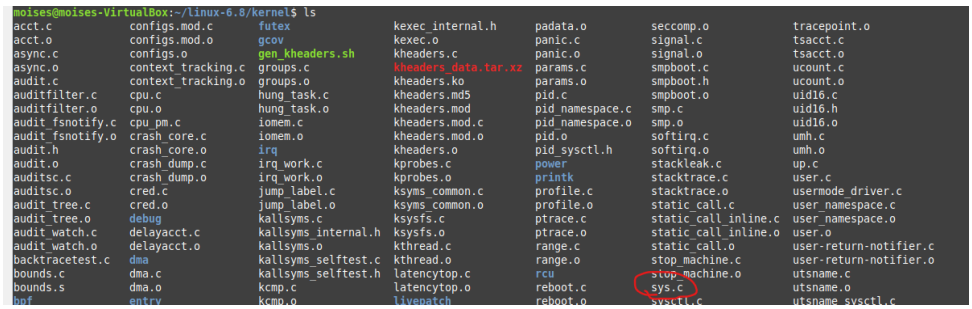
Cada arquitectura correspondiente tiene su propia configuración de syscalls por eso el arch/x86/entry/sycalls. “Syscall_64.tbl” define la **tabla de syscalls** para sistemas de 64 bits en la arquitectura x86.

Cada entrada en esta tabla mapea con número de syscall, nombre de la función asociada y función a ejecutar.



La segunda columna indica las convenciones de llamada a nivel de máquina para el controlador de llamadas del sistema, es decir, cómo se pasan los parámetros. Se denomina *ABI o interfaz binaria de aplicación*. Para nuestra configuración de 64 bits, common es el ABI habitual.

2. **Escribir el código de la función en sys.c de la siguiente ruta.**



En este archivo se ve cómo se definen las llamadas al sistema (syscalls) en el código fuente del Kernel. Es aquí donde podemos crear el código de nuestras propias funciones, tomar en cuenta si es necesario importar librerías correctamente.

3. CODIGO DE PRIMERA FUNCION Y MODIFICACIONES

```
#include <linux/export.h>
#include <linux/mm.h>           //Acceso a la memoria
#include <linux/mm_inline.h>
```

```
#include <linux/uaccess.h> // Para copiar al usuario
#include <asm/io.h>
#include <asm/unistd.h>

#include "uid16.h"

//---------- IMPORTACIONES ------------
#include <linux/mymemorySys.h> //Para encabezado de mi struct y sea accesible
#include <linux/mmzone.h> // Para funciones de memoria
#include <linux/swap.h> //Para informacion de swap
#include <linux/vmstat.h> //Estado de memoria virtual
```

Estas librerías son las necesarias para esta llamada, algunas ya están incluidas y otras deben agregarse. Se debe agregar el archivo ".h" con la estructura que espera recibir la llamada.

```
SYSCALL_DEFINE1(moises_tamalloc, size_t, size) {
    unsigned long addr;

    // Validar un size valido
    if (size == 0 || size > TASK_SIZE) {
        return -EINVAL;
    }

    // Reservar memoria virtual sin asignar páginas físicas (MAP_NORESERVE)
        unsigned long populate = 0;  // Inicializa como 0 para el puntero
        vm_flags_t vm_flags = 0;     // No configuraciones especiales
        unsigned long pgoff = 0;     // Desplazamiento de página en 0
        struct list_head *uf = NULL; // No se usa lista entonces NULL

        addr = do_mmap(NULL, 0, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, vm_flags, pgoff, &populate, uf);

    if (IS_ERR_VALUE(addr)) {
        return addr; // Retorna el error si do_mmap falla
    }

    // Retornar dirección de memoria virtual reservada
    return addr;
}
```

Código de función:
Define una nueva llamada al sistema en el kernel de Linux, moises_tamalloc, para asignar memoria virtual al proceso del usuario. A continuación, se explican los detalles de cada sección del código:

- SYSCALL_DEFINE1: Macro que define una syscall con un argumento. En este caso, la syscall se llama moises_tamalloc y toma un argumento size de tipo size_t, que representa el tamaño de memoria solicitado por el usuario en bytes.

- El objetivo de esta syscall es reservar un área de memoria virtual para el proceso llamante.

- Validación del argumento size
  if (size == 0 || size > TASK_SIZE) {
  return -EINVAL;
  }
  - size == 0: No tiene sentido reservar un bloque de memoria de tamaño cero. Se considera un error.
  - size > TASK_SIZE: En Linux, TASK_SIZE es el límite superior para la memoria accesible por un proceso de usuario. Si el tamaño solicitado excede este límite, también es inválido.
  - return -EINVAL: Si el tamaño es inválido, la syscall retorna -EINVAL, indicando un error de argumento inválido.

- Configuración para reservar memoria virtual
  unsigned long populate = 0;
  vm_flags_t vm_flags = 0;
  unsigned long pgoff = 0;
  struct list_head *uf = NULL;

  Esto debido a la struct que podemos encontrar en mm/mmap.c

```
*/
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long len, unsigned long prot,
                      unsigned long flags, vm_flags_t vm_flags,
                      unsigned long pgoff, unsigned long *populate,
                      struct list_head *uf)
{
```

  - populate: Controla si se deben poblar las páginas físicas de inmediato. Aquí está inicializado a 0, lo que indica que las páginas no se asignan físicamente todavía.
  - vm_flags: Bandera para configuraciones especiales en el área virtual. En este caso, no se usa ninguna configuración personalizada.
  - pgoff: Es el desplazamiento de página. Inicializado en 0, indica que no se aplica ningún desplazamiento.
  - uf: Es un puntero a una lista que se puede usar para información adicional del área de memoria. Aquí se inicializa como NULL, ya que no se necesita.

- Llamada a do_mmap
  addr = do_mmap(NULL, 0, size, PROT_READ | PROT_WRITE,
  MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, vm_flags, pgoff, &populate, uf);

  do_mmap: Función del kernel para crear un área de memoria virtual.
  - NULL: Indica que no se especifica una dirección preferida para el mapeo.
  - 0: Dirección base inicial (el kernel decide dónde asignarla).
  - size: Tamaño solicitado para el área de memoria.
  - PROT_READ | PROT_WRITE: Permite que el área de memoria se pueda leer y escribir.
  - MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE:
    - MAP_PRIVATE: El área de memoria es privada al proceso.
    - MAP_ANONYMOUS: No está asociada a ningún archivo.
    - MAP_NORESERVE: No se reservan páginas físicas inmediatamente, lo que ahorra recursos.
  - vm_flags: Aquí no se aplica ninguna configuración adicional.
  - pgoff: Desplazamiento de página (0 en este caso).
  - &populate: Dirección donde se almacenará la información de "población" de páginas.
  - uf: Lista de información adicional (no se usa).

  El resultado de do_mmap es:
  - Una dirección base virtual válida si tiene éxito.
  - Un valor negativo si ocurre un error.

Manejo de errores de do_mmap

```
if (IS_ERR_VALUE(addr)) {
  return addr;
}
```

IS_ERR_VALUE: Macro que verifica si la dirección retornada por do_mmap es un código de error.

Si hay un error, el valor de error se retorna directamente al usuario.

Retorno de la dirección virtual

```
return addr;
```

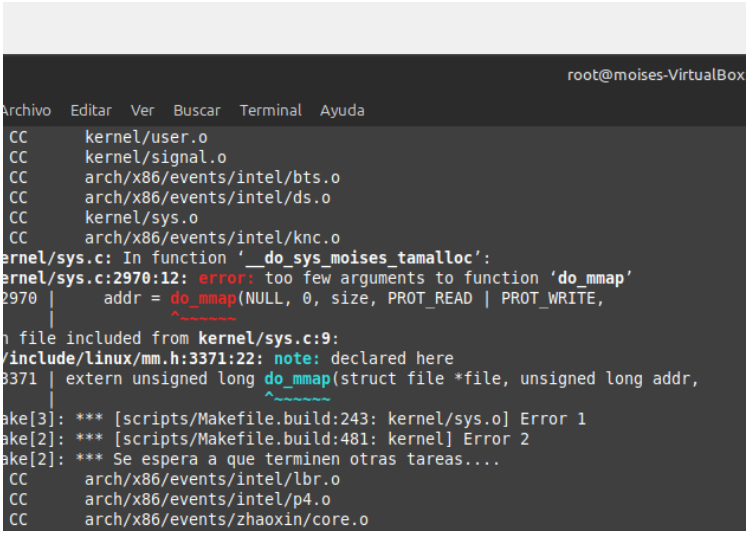
- Si no hubo errores, la syscall retorna la dirección base del área de memoria reservada.
- Esta dirección es válida en el espacio de usuario y puede ser utilizada por el proceso para leer/escribir datos.

Importancia de este diseño

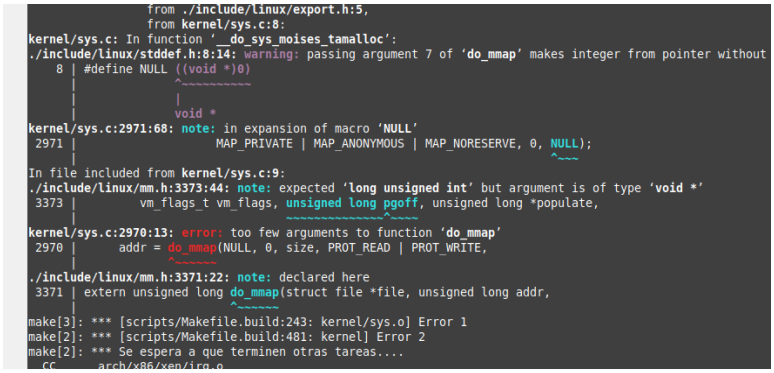
1. Validación estricta de argumentos: Garantiza que no se realicen solicitudes inválidas, protegiendo al kernel.
2. Uso eficiente de recursos: La memoria física no se reserva hasta que realmente sea necesario (MAP_NORESERVE).
3. Flexibilidad: do_mmap permite asignar memoria con características configurables según las necesidades del proceso.
4. Seguridad: El uso de macros como IS_ERR_VALUE evita accesos inseguros a direcciones inválidas.
   Este enfoque es eficiente y robusto, y sigue las mejores prácticas del desarrollo en el kernel de Linux.

Pruebas y errores encontrados

El primer error ocurre porque no se le estaban pasando demasiados argumentos, esto por la falta de conocimiento sobre la estructura que es estaba invocando.



El segundo error fue por el tipo de dato de los argumentos, los cuales no eran los correctos.



SOLUCIONES:

Para solucionar los errores antes mencionado se utilizó la documentación sobre el Kernel para saber cuales eran los parámetros correctos y su tipo de dato, de tal manera que se crearon variables para satisfacer los requerimientos:

```
unsigned long populate = 0;
vm_flags_t vm_flags = 0;
unsigned long pgoff = 0;
struct list_head *uf = NULL;
```

MODULOS

Se crearon 3 módulo para captura las métricas necesarias utilizando las llamadas personalizadas y mostrando en consola para análisis de la información y monitoreo. La información es capturada y guardada en la carpeta /proc con archivos de distintos nombres.

Comandos generales para interactuar con este apartado:

- `Instalar módulo: sudo insmod moduleSys.ko`
- `Remover módulo: sudo rmmod moduleSys`
- `Ver contenido: cat /proc/[name]`

La información recopilada es la siguiente:

- Para CADA proceso individual:
  - Memoria reservada (Reserved) (en KB/MB)
  - Memoria utilizada (Commited) (en KB/MB, y en % de memoria reservada)
  - OOM SCore
- En resumen de TODOS los procesos:
  - Memoria total reservada (Reserved) (en MB)
  - Memoria total utilizada (Commited) (en MB)

El Makefile para todos es de la misma estructura con diferencia en el nombre de creación, se dan instrucciones para comando make y make clean:

```
obj-m += moduleMyCPU.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Encabezados: También es la misma configuración en donde cambia la descripción y el nombre del archivo creado en /proc. Se tienen librerías para escritura y acceso a carpetas específicas.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Moises");
MODULE_DESCRIPTION("Modulo para mostrar el uso de la CPU");
MODULE_VERSION("1.0");

#define PROC_NAME "cpu_usage"
```

Se tiene también el mismo flujo para la creación del archivo en /proc y su respectiva lectura, también se tienen los método de carga y descarga del módulo.

```
static int cpu_usage_open(struct inode *inode, struct file *file) {
    return single_open(file, cpu_usage_show, NULL);
}

static const struct proc_ops cpu_usage_ops = {
    .proc_open = cpu_usage_open,
    .proc_read = seq_read,
};

static int __init cpu_usage_init(void) {
    proc_create(PROC_NAME, 0, NULL, &cpu_usage_ops);
    printk(KERN_INFO "CPU usage module loaded\n");
    return 0;
}

static void __exit cpu_usage_exit(void) {
    remove_proc_entry(PROC_NAME, NULL);
    printk(KERN_INFO "CPU usage module unloaded\n");
}

module_init(cpu_usage_init);
module_exit(cpu_usage_exit);
```

1. Modulo Syscall

```c
#define SYS_MOISES_TAMALLOC 551

int main() {

    printf("Programa para tamalloc PID: %d\n", getpid());
    printf("Este programa aloja memoria con tamalloc. ENTER para continuar...\n");
    getchar();

    size_t total_size = 10 * 1024 * 1024; // 10 MB

    // Llamada a la syscall
    char *buffer = (char *)syscall(SYS_MOISES_TAMALLOC, total_size);

    if ((long)buffer < 0) {
        perror("tamalloc error");
        return 1;
    }

    printf("Se alojan 10MB de memoria en la direccion: %p\n", buffer);
    printf("ENTER para leer memoria byte por byte...\n");
    getchar();
```

```c
// Lee la memoria byte por byte para verificar que fue inicializada a cero
for (size_t i = 0; i < total_size; i++) {
    char t = buffer[i]; // Accede al byte en la posición i, esto activa la asignación perezosa (Lazy

    if (t != 0) { // Si no es cero, significa que no se inicializó correctamente
        printf("ERROR FATAL: Memoria en byte %zu no fue inicializada en 0\n", i);
        return 10;
    }

    // Escribe una letra aleatoria A-Z para activar la técnica Copy-on-Write (CoW)
    char random_letter = 'A' + (rand() % 26);
    buffer[i] = random_letter;

    // Cada 1 MB, imprime el progreso y espera 1 segundo
    if (i % (1024 * 1024) == 0 && i > 0) {
        printf("Revisado %zu MB...\n", i / (1024 * 1024));
        sleep(2);
    }
}
```

Realmente no es un módulo cargable, si no un programa en el espacio de usuario.

Se define el número de llamada asignada. Son varios procedimientos.

Primero se obtiene el PID actual y se preparan 10 MB para alojar utilizando la syscall creada. Si todo está bien se obtiene la dirección de memoria que se utiliza.

Luego se tiene el recorrido de 1MB para ir accediendo y verificar que la memoria se inicialice en cero, de lo contrario se notificará.

Se envía la lectura/escritura de un carácter aleatorio cada 2 segundos.

2. Verificar proceso

```c
static int watch_pid = -1; // PID a monitorear (-1 para ninguno)

// Funcion para mostrar informacion del PID
static int watch_pid_usage_show(struct seq_file *m, void *v) {

    struct task_struct *task;
    struct mm_struct *mm;

    if (watch_pid == -1) {
        seq_printf(m, "{ \"error\": \"No se ha configurado un PID.\" }\n");
        return 0;
    }

    // Buscar el proceso por PID
    rcu_read_lock(); // Protege las operaciones de lectura en estructuras del kernel
    task = pid_task(find_vpid(watch_pid), PIDTYPE_PID);
    if (!task) {
        rcu_read_unlock();
        seq_printf(m, "{ \"error\": \"PID no válido o proceso no encontrado.\" }\n");
        return 0;
    }
```

```c
    // Acceder a la memoria del proceso
    mm = task->mm;
    if (!mm) {
        rcu_read_unlock();
        seq_printf(m, "{ \"error\": \"El proceso no tiene espacio de memoria asignado.\" }\n");
        return 0;
    }

    // Extraer estadísticas de memoria
    // Calcular VmSize y VmRSS
    unsigned long vmsize = mm->total_vm << (PAGE_SHIFT - 10); // Convertir de páginas a KB
    unsigned long vmrss = get_mm_rss(task->mm) * (PAGE_SIZE/1024); // Páginas residenciales a KB

    seq_printf(m, "{\n");
    seq_printf(m, "  \"VmSize\": \"%lu kB\",\n",vmsize );
    seq_printf(m, "  \"VmRSS\": \"%lu kB\",\n", vmrss);
    seq_printf(m, "  \"OOM_Score\": \"%d\"\n", task->signal->oom_score_adj);
    seq_printf(m, "}\n");

    rcu_read_unlock();
    return 0;
}
```

Resumen del flujo de operaciones

1. **Carga del módulo: Se crea el archivo /proc/watch_202010833.**
2. **Escritura en el archivo (echo <PID> > /proc/watch_202010833): Cambia el PID que se monitorea.**
3. **Lectura del archivo (cat /proc/watch_202010833): Devuelve estadísticas del proceso o errores en caso de problemas.**
4. **Descarga del módulo: Limpia la entrada en /proc y libera recursos.**

Si el PID es válido, se obtienen y devuelven las siguientes estadísticas en formato JSON:

VmSize: Tamaño total de memoria virtual del proceso en kilobytes.

VmRSS: Tamaño de la memoria física utilizada por el proceso en kilobytes.

OOM_Score: Puntaje de ajuste de eliminación por falta de memoria (oom_score_adj).

3. Modulo de estadísticas

```c
// Estructura para almacenar la información del proceso
struct proc_stats {
    pid_t pid;
    char name[TASK_COMM_LEN];  // Nombre del proceso
    unsigned long vm_size;   // Memoria reservada (VmSize)
    unsigned long vm_rss;    // Memoria utilizada (VmRSS)
    int oom_score;    // OOM Score
};

// Función para obtener las estadísticas de un proceso específico
static int get_proc_stats(struct task_struct *task, struct proc_stats *stats) {
    struct mm_struct *mm;

    mm = task->mm;
    if (!mm) return -1;

    // Copiar el nombre del proceso y  calcular VmSize y VmRSS
    strncpy(stats->name, task->comm, TASK_COMM_LEN);
    stats->vm_size = mm->total_vm << (PAGE_SHIFT - 10); // Convertir de páginas a KB
    stats->vm_rss = get_mm_rss(task->mm) * (PAGE_SIZE/1024); // Páginas residenciales a KB
```

```c
// Función para mostrar estadísticas en formato JSON
static int procs_stats_show(struct seq_file *m, void *v) {
    struct task_struct *task;
    struct proc_stats stats;
    unsigned long total_vm_size = 0, total_vm_rss = 0;
    int total_procs = 0;

    seq_printf(m, "{\n");
    seq_printf(m, "  \"procs\": [\n");

    rcu_read_lock();
    for_each_process(task) {
        if (get_proc_stats(task, &stats) == 0) {
            // Mostrar estadísticas del proceso en formato JSON
            if (total_procs > 0) seq_printf(m, ",\n");
            seq_printf(m, "    {\n");
            seq_printf(m, "      \"pid\": %d,\n", stats.pid);
            seq_printf(m, "      \"name\": \"%s\",\n", stats.name);
            seq_printf(m, "      \"vm_size (KB)\": %lu,\n", stats.vm_size);
            seq_printf(m, "      \"vm_rss (KB)\": %lu,\n", stats.vm_rss);
            seq_printf(m, "      \"oom_score\": %d\n", stats.oom_score);
            seq_printf(m, "    }");

            total_vm_size += stats.vm_size;
            total_vm_rss += stats.vm_rss;
            total_procs++;
        }
```

Características principales:

1. **Información por proceso:**
   - **pid: ID del proceso.**
   - **name: Nombre del proceso.**
   - **vm_size: Tamaño total de memoria virtual reservada (VmSize) en KB.**
   - **vm_rss: Tamaño de memoria residente (VmRSS) en KB.**
   - **oom_score: Puntaje de OOM (Out-of-Memory) del proceso.**
2. **Resumen global:**
   - **total_vm_size: Suma de la memoria virtual reservada por todos los procesos.**
   - **total_vm_rss: Suma de la memoria residente de todos los procesos.**
   - **total_procs: Número total de procesos.**
3. **Salida en formato JSON**

Funciones principales:

1. **get_proc_stats:**

Recopila información de un proceso específico, accediendo a su estructura task_struct y mm_struct para obtener las estadísticas de memoria y el nombre del proceso.

2. **procs_stats_show:**

Itera sobre todos los procesos del sistema usando for_each_process.

Llama a get_proc_stats para obtener información de cada proceso válido.

Formatea la información en JSON y la imprime en el archivo /proc/statsAllProcs.

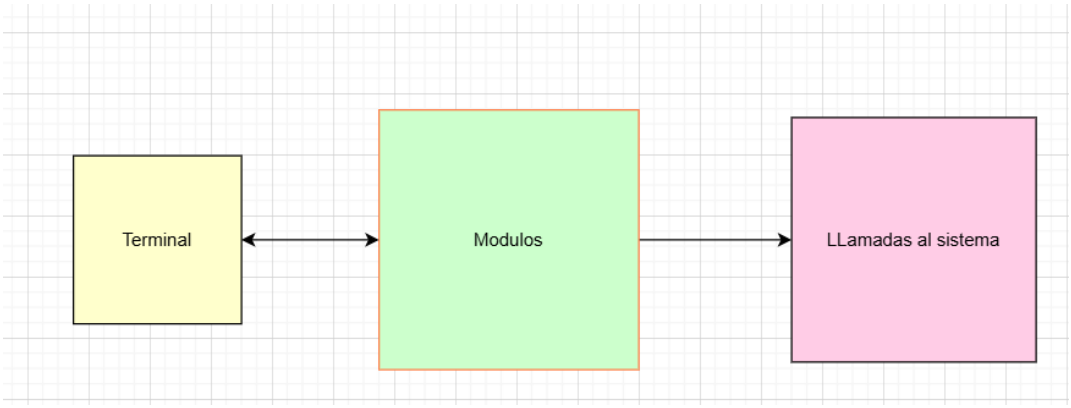
Python

```python
        return None, None

# Llamar a la función para obtener los datos
df, global_data = parse_stats_file(file_path)

# Estilización adicional: usar tabulate para la tabla
if df is not None:
    # Calcular valores en MB y porcentaje de uso para los procesos
    df['vm_size (MB)'] = df['vm_size (KB)'] / 1024
    df['vm_rss (MB)'] = df['vm_rss (KB)'] / 1024
    df['usage (%)'] = (df['vm_rss (KB)'] / df['vm_size (KB)']) * 100

    # Estilizar datos de procesos
    df['oom_score'] = df['oom_score'].apply(lambda x: colored(x, 'red') if x <= 0 else str(x))
    df['pid'] = df['pid'].apply(lambda x: colored(x, 'green'))
    df['name'] = df['name'].apply(lambda x: colored(x, 'cyan'))
    df['vm_size (KB)'] = df['vm_size (KB)'].apply(lambda x: colored(f"{x} KB", 'yellow'))
    df['vm_rss (KB)'] = df['vm_rss (KB)'].apply(lambda x: colored(f"{x} KB", 'magenta'))
    df['vm_size (MB)'] = df['vm_size (MB)'].apply(lambda x: colored(f"{x:.2f} MB", 'yellow'))
    df['vm_rss (MB)'] = df['vm_rss (MB)'].apply(lambda x: colored(f"{x:.2f} MB", 'magenta'))
    df['usage (%)'] = df['usage (%)'].apply(lambda x: colored(f"{x:.2f}%", 'blue'))

    # Usar tabulate para una presentación más bonita
    table = tabulate(df, headers='keys', tablefmt='fancy_grid', showindex=False)

    # Mostrar la tabla estilizada
    print("Datos de Procesos:")
```

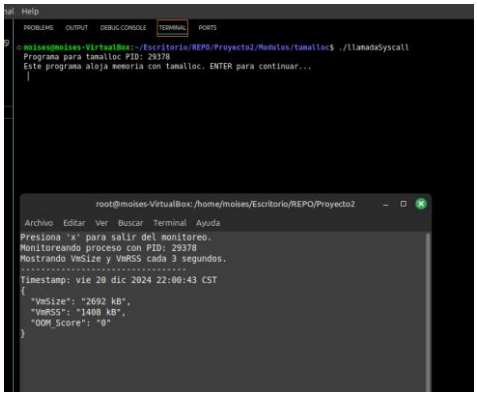
Para ver las estadísticas estilizadas se usa Python para leer los archivos creados por los módulos y mostrar la información en tablas de colores y ordenadas.

Se usó pandas, tabulate y termcolor para los estilos.

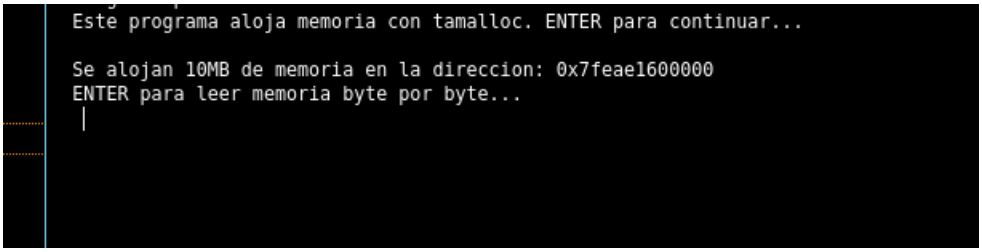
ESQUEMA



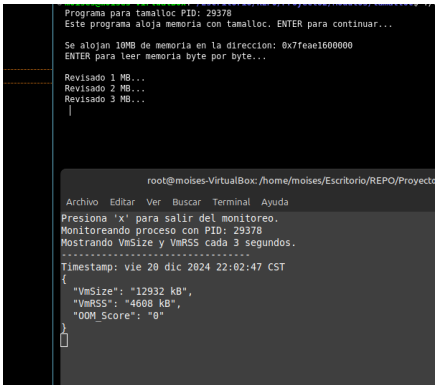
USO DEL SISTEMA



Primero se ejecuta ./llamadaSyscall y miScript.sh.
El script solicita el PID a monitorear y se mantiene así hasta presionar la tecla indicada. Luego de presionar Enter comienza la asignación.



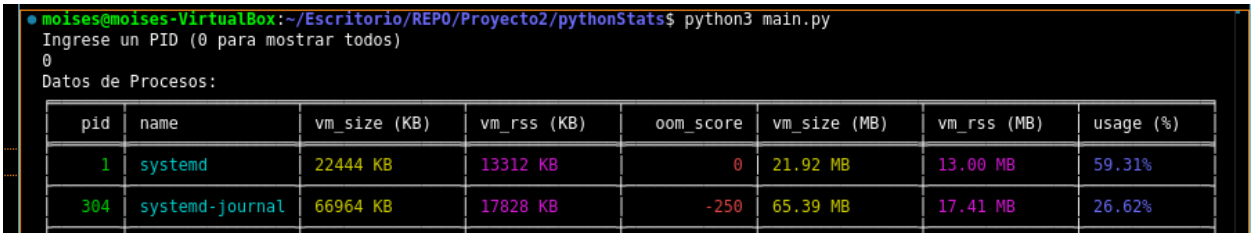
Presione enter para iniciar el procedimiento:

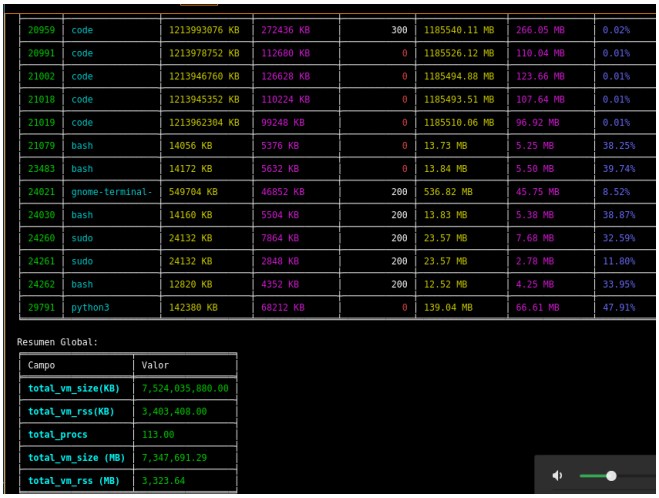


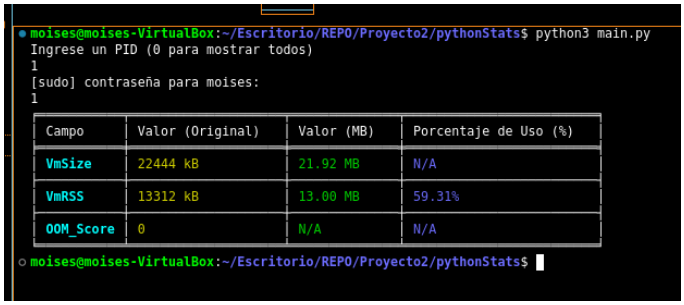
Note como cambian los valores hasta que se accede a la dirección reservada.

Estadísticas

Puede indicar un pid en específico o 0 para mostrar información de todos los procesos.







Repositorio: https://github.com/Hrafnyr/SO2_202010833_VD2024/tree/main/Proyecto2

CRONOGRAMA:

16/12/24 – 20/12/24

LUNES	MARTES	MIERCOLES	JUEVES	VIERNES	SABADO
Documentarse	Syscall tamalloc	Pruebas de asignación	Script de pruebas y módulo 1	Módulo 2	
Documentarse				Modulo 3	
		Testing inicial		Estilización de resultados	
				Documentacion	
				Entrega de proyecto	

Análisis de resultados:

VmSize (Virtual Memory Size):

- Representa la memoria virtual total reservada por el proceso, incluidos segmentos como código, datos, pila, memoria mapeada (incluidos archivos) y memoria compartida.
- Incluso un proceso que no ha realizado muchas operaciones tendrá segmentos predefinidos, como el espacio para el código ejecutable, bibliotecas cargadas y la pila.
- Por eso, VmSize nunca empieza en 0, ya que el sistema operativo asigna memoria al proceso durante su inicialización.

VmRSS (Resident Set Size):

- Es la parte de VmSize que está realmente cargada en RAM.
- Aunque el proceso reserve memoria virtual (VmSize), no toda esta memoria se mapea a RAM inmediatamente. Solo las páginas que el proceso realmente utiliza se cargan en memoria física.
- VmRSS no comienza en 0 porque, al inicio, algunas páginas como las de código y estructuras iniciales del proceso ya están en RAM.

¿Por qué VmRSS aumenta al acceder a memoria?

Por cada acceso a una página nueva, se genera una falta de página (page fault), y el kernel asigna esa página a memoria física. Esto incrementa VmRSS.

¿Por qué VmRSS no alcanza a VmSize al finalizar?

- **Reserva vs. Uso Real**:
    - VmSize incluye todo el espacio reservado, incluso si nunca se utiliza (por ejemplo, memoria reservada para la pila o grandes asignaciones con malloc que no se escriben).
    - Solo las páginas que son accedidas por el proceso se convierten en parte de VmRSS.
    - VmSize es siempre igual o mayor que VmRSS porque representa la memoria total reservada, mientras que VmRSS es solo la parte activa en RAM.

CONCLUSIONES

En este proyecto se logró implementar, analizar y depurar módulos del kernel que interactúan con el sistema mediante nuevas *syscalls* y archivos en /proc. Este esfuerzo ha permitido desarrollar herramientas personalizadas para monitorear aspectos críticos como el uso de la CPU y la memoria, utilizando funcionalidades avanzadas del kernel de Linux.

Ahora se aprende sobre el manejo de asignación de memoria y el uso solamente cuando se accede a la sección. Reservar una región de memoria virtual de tamaño size para procesos de usuario, permite su uso posterior mediante asignaciones de páginas físicas cuando sean requeridas.

Se logró comprender y manipular directamente la asignación y el uso de memoria en el kernel de Linux. Esto permitió observar cómo el kernel gestiona los recursos de memoria virtual y física, así como la relación entre ellos.