



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 1

Nombre

Moises David Maldonado de León

Carné

202010833

Guatemala, 14 de diciembre de 2024

Introducción

En el presente documento, se describe la lógica, estructura y los demás aspectos técnicos del proyecto 1 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 implementan tres nuevas llamadas al sistema que son accesibles 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, disco o cantidad de veces que se invoca una llamada en específico.

OBJETIVOS

Generales

Familiarizar al lector con la lógica planteada para el funcionamiento del sistema realizado mediante el uso del lenguaje de programación C y documentación del Kernel así como 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.

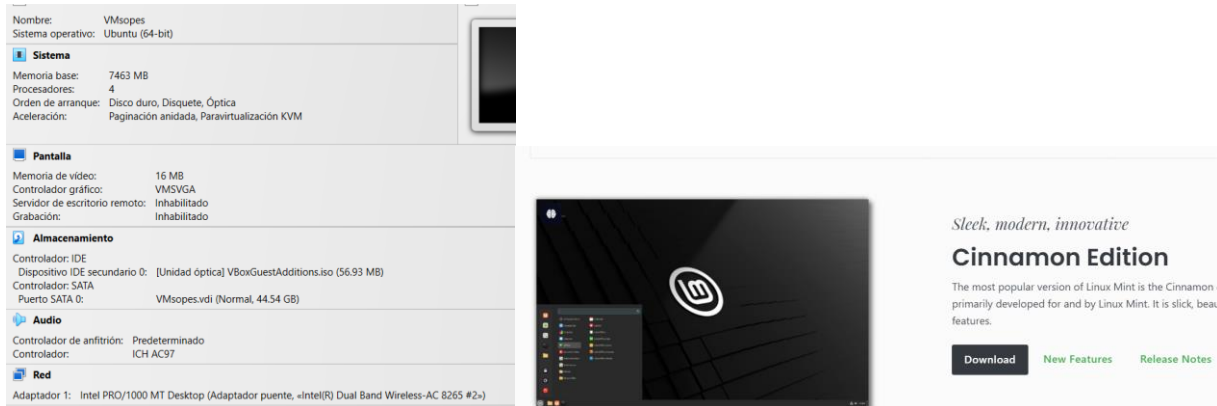
Alcances

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.

Lógica de programación

Configuración del entorno

Para no entrar en conflicto con el kernel actual se descarga uno para poder trabajar con otra versión sin comprometer el nuestro. En este caso se utiliza una máquina virtual con la distribución de mediante VirtualBox con las siguientes características:



Guía inicial de configuración:

Paso 1.

Descargar version de kernel a utilizar:

<https://www.kernel.org/pub/linux/kernel/v6.x/linux-6.8.tar.xz>

Con ls puede ver que si tenga el archivo linux-6.8.tar.xz

Paso 2.

Descomprimir

```
tar -xf linux-6.8.tar.xz
```

Esto creará una carpeta con el nombre (sin extensión .tar.xz), puede ver con: ls -al

Como extra para hacer CD a la carpeta vamos a mover con _ en el nombre:

```
mv linux-6.8.tar.xz _linux-6.8.tar.xz
```

Paso 3.

Ahora cd linux-6.8 para acceder a la carpeta

Instalaciones iniciales:

Puede dar un error de que kernel-package no existe. De ser así, solo quitar de la lista:

```
sudo apt-get install build-essential kernel-package libncurses5-dev fakeroot wget bzip2 openssl
```

```
sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev
```

Copiar el archivo de config por primera vez, hay uno por cada kernel:

```
cp -v /boot/config-$(uname -r) .config
```

Posible error:

```
make[3]: *** No rule to make target 'debian/canonical-certs.pem', needed by  
'certs/x509_certificate_list'. Stop.
```

Ejecutar esto en la raíz del proyecto para crear nuevos certificados, que es el error que indica:

```
scripts/config --disable SYSTEM_TRUSTED_KEYS
```

```
scripts/config --disable SYSTEM_REVOCATION_KEYS
```

Cambiar para que no se sobrescriba el kernel actual:

```
nano Makefile
```

Esto abrirá un archivo de configuración, en “extraversión”

podemos colocar información que saldrá con el comando `uname -r`

Paso 4:

Ejecutar script de compilación teniendo la consola como un superusuario:

```
sudo -s
```

Notas sobre las configuraciones:

Make clean para limpiar archivos creados

Opciones:

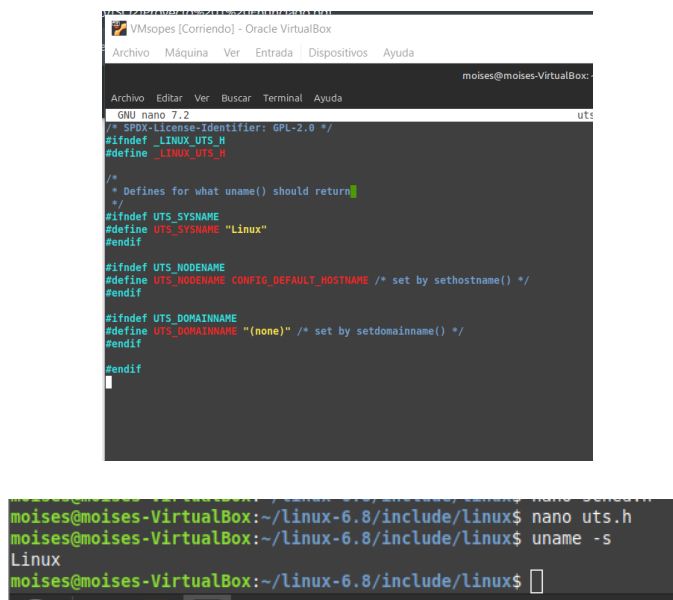
- **make oldconfig:** Toma el config actual y si la versión tiene características que este no tiene, pregunta si queremos usarla.
- **make menuconfig:** Manera grafica de ver algunas preguntas del script, como tipo menú de BIOS.
- **make localmodconfig:** Recomendado de proceso activo de desarrollo, esto solo compila los módulos cargados actuales en el sistema.

Modificaciones en el kernel

Se va a agregar un nombre personalizado, así como un mensaje de bienvenida al iniciar el sistema.

Nombre personalizado

1. En la carpeta del Kernel ir a la carpeta include: `cd include/`
2. Esta carpeta contiene los encabezados que definen estructuras, macros y funciones importantes del Kernel. Entre estos importantes podemos mencionar 3: el `sched.h` con el `task_struct` para ver los procesos, así como información de memoria; el `uapi`, directorio donde se define la API del espacio de usuario y finalmente el `uts.h`, donde tiene la información del nombre:



```
moises@moises-VirtualBox:~/linux-6.8/include/linux$ nano uts.h
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_UTS_H
#define _LINUX_UTS_H

/*
 * Defines for what uname() should return
 */
#ifndef UTS_SYSNAME
#define UTS_SYSNAME "Linux"
#endif

#ifndef UTS_NODENAME
#define UTS_NODENAME CONFIG_DEFAULT_HOSTNAME /* set by sethostname() */
#endif

#ifndef UTS_DOMAINNAME
#define UTS_DOMAINNAME "(none)" /* set by setdomainname() */
#endif
#endif

moises@moises-VirtualBox:~/linux-6.8/include/linux$ uname -s
Linux
moises@moises-VirtualBox:~/linux-6.8/include/linux$
```

Se cambiará UTS_SYSNAME "Linux" a UTS_SYSNAME "Linux ".

Mensaje personalizado

1. En la carpeta del Kernel ir a la carpeta init: `cd init/`
Esta carpeta contiene archivos y código responsables de inicializar el sistema una vez que el kernel ha sido cargado.
2. Ahora modificamos el `main.c` alojado en esta carpeta. Este es el archivo principal que contiene la función **start_kernel**, que es el punto de entrada principal del Kernel después de la carga por el bootloader. Aquí se configuran aspectos fundamentales como: Configuración de memoria, Inicialización de subsistemas básicos (CPU, timers), Montaje del sistema de archivos raíz.
3. Vamos a modificar el código de `start_kernel()` justo al inicio.
Se usa `printk()` para mostrar un mensaje y el nivel de prioridad en este caso, `KERN_INFO` que es para información general, es decir, no representa un error, advertencia o mensaje crítico..

[illegible]

```

moises@moises-VirtualBox: ~$ sudo ./initfs_dmesg
0 00000000 Linux version 6.8.0-49-ucsl (root@moises-VirtualBox) (gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0, GNU ld (GNU Binutils for Ubuntu) 2.42) #2 SMP PRE
MPT_DYNAMIC Thu Dec 5 22:11:21 CST 2024
0 00000000 Command line: BOOT_IMAGE=/boot/vmlinuz-6.8.0-49-ucsl root=UUID=27561263-414b-4e5c-84d4-bac64626d2e1 ro quiet splash
0 00000000 KERNEL supported cpus:
0 00000000 Intel GenuineIntel
0 00000000 AMD AuthenticAMD
0 00000000 Hygon HygonGenuine
0 00000000 Centaur CentaurHauls
0 00000000 zhaoxin Shanghai

```

Para ver reflejados los cambios se compila el Kernel:

[illegible]

Nuevas llamadas al sistema

capture_memory_snapshot

Flujo:

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

```
moises@moises-VirtualBox:~/linux-6.8/arch/x86/entry/syscalls$ ls
Makefile  syscall_32.tbl  syscall_64.tbl
moises@moises-VirtualBox:~/linux-6.8/arch/x86/entry/syscalls$
```

Cada arquitectura correspondiente tiene su propia configuración de syscalls por eso el arch/x86/entry/syscalls. "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.

426	545 x32 execveat	compat_sys_execveat
427	546 x32 preadv2	compat_sys_preadv64v2
428	547 x32 pwritev2	compat_sys_pwritev64v2
429	# This is the end of the legacy x32 range. Numbers 548 and above are	
430	# not special and are not to be used for x32-specific syscalls.	
431	548 common capture_memory_snapshot	sys_capture_memory_snapshot

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.

```
moises@moises-VirtualBox:~/linux-6.8/kernel$ ls
acct.c      configs.mod.c  futex          kexec_internal.h  padata.o      seccomp.o      tracepoint.o
acct.o      configs.mod.o  gcov           kexec.o            panic.c       signal.c       tsacct.c
asyncc.c   context_tracking.c  gen_kheaders.sh  kheaders.c         panic.o       signal.o       tsacct.o
asyncc.o   context_tracking.o  groups.o         kheaders.data.tar.xz  params.c      smboot.c      ucount.c
audit.c    cpu.c           hung_task.c     kheaders.md5       pid.c         smboot.h      uid16.c
auditfilter.c  cpu.o          iomem.c        kheaders.mod.c     pid_namespace.c  smp.c         uid16.h
auditfsnotify.c  crash_core.c  irq             kheaders.mod.o     pid_namespace.o  smp.o         uid16.o
auditfsnotify.o crash_core.o   irq_work.c     kheaders.o         pid.o          softirq.c     umh.c
audit.h     crash_dump.c    irq_work.o     kprobes.c          pid_sysctl.h    softirq.o     umh.o
audit.o     crash_dump.o   jump_label.c   kprobes.o          power          stackleak.c   up.c
auditsc.c   cred.c          jump_label.o   kprobes.o          printk         stacktrace.c  user.c
auditsc.o   cred.o          kallsyms.c     ksysfs.o           profile.c       stacktrace.o  usermode_driver.c
audit_tree.c  debug           kallsyms.o     ksysfs.o           profile.o       static_call.c  user_namespace.c
audit_tree.o delayacct.c     kallsyms.o     ksysfs.o           ptrace.c       static_call.inline.c  user_namespace.o
audit_watch.c delayacct.o     kallsyms.o     kthread.c          ptrace.o       static_call.inline.o  user.o
audit_watch.o dma.c           kallsyms_selftest.c  kthread.o       range.c        stop_machine.c  user-return-notifier.c
backtracet.c dma.o           kallsyms_selftest.h  latencytop.c    range.o        stop_machine.o  user-return-notifier.o
bounds.c    dma.o           kcmp.o         latencytop.o       reboot.c       sys.c         utstime.o
bounds.o    entry           kcmp.o         livepatch           reboot.o       sysctl.c      utstime_sysctl.c
```

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(capture_memory_snapshot, struct my_memory_snapshot __user *, user_snapshot)
{
    struct my_memory_snapshot snapshot;
    struct sysinfo si; //Estructura para obtener informacion del sistema

    //Llenar estructura con metricas, en el modulo se pasa a KB
    snapshot.memoria_cache = global_node_page_state(NR_FILE_PAGES);
    snapshot.paginas_activas = global_node_page_state(NR_ACTIVE_FILE);
    snapshot.paginas_inactivas = global_node_page_state(NR_INACTIVE_FILE);
    snapshot.memoria_sucia = global_node_page_state(NR_FILE_DIRTY);
    snapshot.paginas_en_disco = global_node_page_state(NR_WRITEBACK);

    si.swapinfo(&si);
    snapshot.swap_total = si.totalswap;
    snapshot.swap_libre = si.freeswap;

    if (copy_to_user(user_snapshot, &snapshot, sizeof(snapshot)))
        return -EFAULT;

    return 0; //TODO BIEN
};
```

Código de función:

Se usa define 1 porque se espera un parámetro que será una estructura personalizada creada en “/include/kernel/mymemorySys.h” dado que aquí se tienen los encabezados para ser accedidos desde otro lado.

Se hace referencia al nombre de la llamada personalizada y el struct que recibirá también ya definido; el __user * especifica que el puntero apunta a memoria en el espacio de usuario.

Como se puede observar se tienen los valores necesarios de swap libre y total. Esto a través de la otra estructura `si_swapinfo` que recibe el primer objeto y se puede obtener la información.

```

1 // include /usr/include/sys/types.h
2
3 /* Some Linux identifier: opt > 0 with Linux system, note */
4 #ifndef _LINUX_SYNOPT_H
5 #define _LINUX_SYNOPT_H
6
7 #include <linux/types.h>
8
9 #define SZ_LINUX_SOPT 16
10
11 struct synopt {
12     /* seconds since boot */
13     _kernel_ulong_t loadavg[3];
14     /* %1, %2, and %3 vmstat load averages */
15     _kernel_ulong_t totalram;
16     /* total usable main memory size */
17     _kernel_ulong_t freeram;
18     /* Available memory size */
19     _kernel_ulong_t shrdmem;
20     /* Amount of shared memory */
21     _kernel_ulong_t buffers;
22     /* memory used by buffers */
23     _kernel_ulong_t totalswap;
24     /* total swap space size */
25     _kernel_ulong_t freeswap;
26     /* swap space still available */
27     /* Number of current processes */
28     /* explicit padding for high */
29     _kernel_ulong_t totalhigh;
30     /* total high memory size */
31     _kernel_ulong_t freehigh;
32     /* available high memory size */
33     /* memory unit is bytes */
34     char _[20+sizeof(_kernel_ulong_t)*sizeof(_u64)]; /* padding: lincx uses this. */
35 };
36
37 #endif /* _LINUX_SYNOPT_H */

```

```
void si_swapinfo(struct sysinfo *val)
{
    unsigned int type;
    unsigned long nr_to_be_unused = 0;

    spin_lock(&swap_lock);
    for (type = 0; type < nr_swapfiles; type++) {
        struct swap_info_struct *si = swap_info[type];

        if ((si->flags & SMP_USED) && ! (si->flags & SWP_WRITEOK))
            nr_to_be_unused += READ_ONCE(si->inuse_pages);
    }
    val->freeswap = atomic_long_read(&nr_swap_pages) + nr_to_be_unused;
    val->totalswap = total_swap_pages + nr_to_be_unused;
    spin_unlock(&swap_lock);
}
```

```
static inline unsigned long global_node_page_state(enum node_stat_item item)
{
    VM_WARN_ON_ONCE(vmstat_item_in_bytes(item));

    return global_node_page_state_pages(item);
}
```

```
enum stat_item {
    NR_LRU_BASE,
    NR_INACTIVE_ANON = NR_LRU_BASE, /* must match order of LRU_[IN]ACTIVE */
    NR_ACTIVE_ANON, /* " " " " " */
    NR_INACTIVE_FILE, /* " " " " " */
    NR_ACTIVE_FILE, /* " " " " " */
    NR_UNEVICTABLE, /* " " " " " */
    NR_SLAB_RECLAIMABLE_B,
    NR_SLAB_UNRECLAIMABLE_B,
    NR_ISOLATED_ANON, /* Temporary isolated pages from anon lru */
    NR_ISOLATED_FILE, /* Temporary isolated pages from file lru */
    WORKINGSET_NODES,
    WORKINGSET_REFAULT_BASE,
    WORKINGSET_REFAULT_ANON = WORKINGSET_REFAULT_BASE,
    WORKINGSET_REFAULT_FILE,
    WORKINGSET_ACTIVATE_BASE,
    WORKINGSET_ACTIVATE_ANON = WORKINGSET_ACTIVATE_BASE,
    WORKINGSET_ACTIVATE_FILE,
    WORKINGSET_RESTORE_BASE,
    WORKINGSET_RESTORE_ANON = WORKINGSET_RESTORE_BASE,
    WORKINGSET_RESTORE_FILE,
    WORKINGSET_NODERECLAIM,
    NR_ANON_MAPPED, /* Mapped anonymous pages */
    NR_FILE_MAPPED, /* pagecache pages mapped into pagetables,
    only modified from process context */
    NR_FILE_PAGES,
    NR_FILE_DIRTY,
    NR_WRITEBACK,
    NR_WRITEBACK_TEMP, /* Writeback using temporary buffers */
    NR_SHMEM, /* shmem pages (included tmpfs/GEM pages) */

```

- NR_FILE_PAGES: Páginas asignadas a caché de archivos.
- NR_ACTIVE_PAGES: Páginas activas (accedidas recientemente).
- NR_INACTIVE_PAGES: Páginas inactivas (no accedidas recientemente).
- NR_FILE_DIRTY: Páginas sucias (datos modificados).
- NR_WRITEBACK: Páginas que se están escribiendo.

El resultado devuelto está en **número de páginas**. Para convertirlo a kilobytes, se usa:
Sería así: `global node page state(NR FILE PAGES) << (PAGE SHIFT - 10);`

PAGE_SHIFT es el desplazamiento para calcular el tamaño de una página (normalmente 12 para páginas de 4 KB) y - 10 ajusta el resultado a KB.

Esto porque el desplazamiento representado será aproximado a 2^{12} y un KB es 2^{10} por lo que el cálculo tradicional será dividir sin embargo de la manera presentada se evita ese cálculo quedando $2^{12} - 2^{10} = 2^2$ y de esa manera se obtiene el cálculo.

Ejemplo para 1000 páginas

$$1000 \ll 2 = 1000 \times (2^2) = 1000 \times 4 = 4000$$

Finalmente, la función copy_to_user es una herramienta proporcionada por el kernel de Linux para copiar datos desde el espacio del kernel al espacio del usuario en la estructura que el usuario va a solicitar.

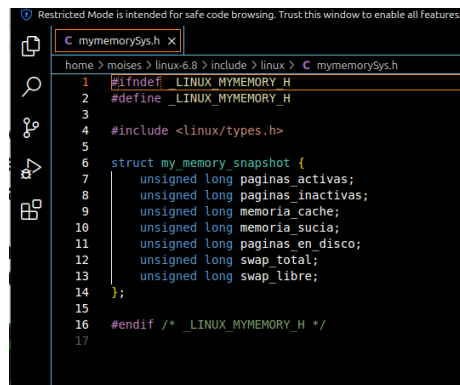
Parámetros:

1. **to:** Dirección de memoria en el espacio del usuario donde se copiarán los datos.
2. **from:** Dirección de memoria en el espacio del kernel que contiene los datos a copiar.
3. **n:** Número de bytes a copiar.

-EFAULT (código de error estándar en Linux) indica que ocurrió un **error de segmentación** o una **dirección de memoria inválida**. Esto se usa para informar que el proceso de usuario proporcionó un puntero incorrecto o inaccesible. De lo contrario se enviará un 0 si todo está bien.

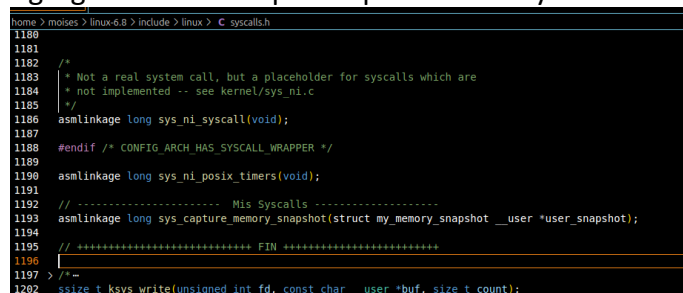
MODIFICACION

En la siguiente ruta se crea un archivo personalizado: mymemorySys.h con el siguiente contenido, esto es debido a que la struct debe ser accesible:



```
1 #ifndef _LINUX_MMEMORY_H
2 #define _LINUX_MMEMORY_H
3
4 #include <linux/types.h>
5
6 struct my_memory_snapshot {
7     unsigned long paginas_activas;
8     unsigned long paginas_inactivas;
9     unsigned long memoria_cache;
10    unsigned long memoria_sucia;
11    unsigned long paginas_en_disco;
12    unsigned long swap_total;
13    unsigned long swap_libre;
14 };
15
16 #endif /* _LINUX_MMEMORY_H */
17
```

Agregue también el prototipo en la ruta y archivo indicado:



```
1188
1189
1190 /*
1191  * Not a real system call, but a placeholder for syscalls which are
1192  * not implemented -- see kernel/sys_ni.c
1193  */
1194 asmlinkage long sys_ni_syscall(void);
1195
1196 #endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
1197
1198 asmlinkage long sys_ni_posix_timers(void);
1199
1200 // ..... Mis Syscalls .....
1201 asmlinkage long sys_capture_memory_snapshot(struct my_memory_snapshot __user *user_snapshot);
1202
1203 // ===== FIN =====
1204
1205 > /*
1206 ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count);
1207
```

En la definición de la syscall se tiene esto:

```
SYSCALL_DEFINE1(capture_memory_snapshot, struct my_memory_snapshot __user *,  
user_snapshot)
```

Por tanto, la estructura del prototipo será utilizando el segundo nombre agregado en la tabla de llamada del sistema (sys_myname) y sus parametros será la struct que se creó y la referencia de que vendrá del espacio de usuario. DEBE AGREGAR la llamada de la estructura en el mismo archivo del prototipo:

```
2 #include <linux/personality.h>  
3 #include <trace/syscall.h>  
4 #include <linux/mymemorySys.h> // SE INCLUYE LLAMADA A MI SYSCALL  
5  
6 #ifdef CONFIG_ARCH_HAS_SYSCALL_WRAPPER  
7 /*
```

A continuación, se muestra un testeo de la salida ya del lado del usuario:

```
moises@moises-VirtualBox: ~/Escritorio  
moises@moises-VirtualBox:~/Escritorio  
Memory Snapshot:  
Cached Memory: 364451 KB  
Active Pages: 234573 KB  
Inactive Pages: 117150 KB  
Dirty Memory: 42 KB  
Writeback Memory: 0 KB  
Swap Total: 524287 KB  
Swap Free: 524287 KB  
moises@moises-VirtualBox:~/Escritorio
```

Pruebas y errores encontrados

Al principio no se creó el archivo mymemorySys.h, solo se definió una struct en el sys.c

```
/* INSTANTANEA DE MEMORIA ESPERADA DEL USUARIO  
struct my_memory_snapshot {  
    unsigned long memoria_cache;  
    unsigned long paginas_activas;  
    unsigned long paginas_inactivas;  
    unsigned long memoria_sucia;  
    unsigned long paginas_en_disco;  
    unsigned long swap_total;  
    unsigned long swap_libre;  
};*/  
  
// *****  
SYSCALL_DEFINE1(capture_memory_snapshot, struct my_memory_snapshot __us  
{  
    struct my_memory_snapshot snapshot;  
    struct sysinfo si; //Estructura para obtener informacion del sistema  
  
    //Llenar estructura con metricas, en el modulo se pasa a KB
```

Y para las métricas se tenía esto:

```
//Llenar estructura con metricas, en el modulo se pasa a KB  
snapshot.memoria_cache = global_node_page_state(NR_FILE_PAGES);  
snapshot.paginas_activas = global_node_page_state(NR_ACTIVE_FILE);  
snapshot.paginas_inactivas = global_node_page_state(NR_INACTIVE_FILE);  
snapshot.memoria_sucia = global_node_page_state(NR_FILE_DIRTY);  
snapshot.paginas_en_disco = global_node_page_state(NR_WRITEBACK);
```

Para paginas activas e inactivas se tenía:

```
warning: 'struct my_memory_snapshot' declared inside parameter list will not be visible
outside of this definition or declaration 1192 | asmlinkage long
sys_capture_memory_snapshot(struct my_memory_snapshot __user *user_snapshot);
| ^~~~~~ kernel/sys.c: In function
'__do_sys_capture_memory_snapshot': kernel/sys.c:2889:59: error:
'NR_ACTIVE_PAGES' undeclared (first use in this function); did you mean
'NR_FILE_PAGES'? 2889 | snapshot.paginas_activas =
global_node_page_state(NR_ACTIVE_PAGES); | ^~~~~~ | NR_FILE_PAGES
kernel/sys.c:2889:59: note: each undeclared identifier is reported only once for each
function it appears in kernel/sys.c:2890:61: error: 'NR_INACTIVE_PAGES' undeclared
(first use in this function); did you mean 'NR_INACTIVE_FILE'? 2890 |
snapshot.paginas_inactivas = global_node_page_state(NR_INACTIVE_PAGES); |
^~~~~~ | NR_INACTIVE_FILE make[3]: *** [scripts/Makefile.build:243:
kernel/sys.o] Error 1 make[2]: *** [scripts/Makefile.build:481: kernel] Error 2 make[2]:
*** Se espera a que terminen otras tareas....
```

Terminando con error de compilación.

SOLUCIONES:

Estructura no visible fuera de la declaración

El compilador no reconoce struct my_memory_snapshot porque no se ha declarado de manera global antes de que sea usada en la declaración de sys_capture_memory_snapshot.

Agregar un archivo .h en /include con la estructura para que sea accesible e importando el mismo en sys.c y syscalls.c

Macros no definidas (NR_ACTIVE_PAGES y NR_INACTIVE_PAGES)

Las macros NR_ACTIVE_PAGES y NR_INACTIVE_PAGES no están disponibles en el contexto actual, lo que puede deberse a una falta de inclusión de encabezados necesarios o a que esas macros han sido reemplazadas por otras.

En este caso se cambian de nombre en PAGES por FILE.

track_syscall_usage

Flujo:

0. Escoger un nombre track_syscall_usage
1. Editar el archivo con la tabla del sistema para agregar la nueva llamada.

```
# syscall_64.tbl
home > moose > linux-6.8 > arch > x86 > entry > syscall > E syscall_64.tbl
423 541 x32 getsockopt sys_getsockopt
424 543 x32 io_setup compat_sys_io_setup
425 544 x32 io_submit compat_sys_io_submit
426 545 x32 execveat compat_sys_execveat
427 546 x32 pread64 compat_sys_pread64
428 547 x32 pwrite64 compat_sys_pwrite64
429 # This is the end of the legacy x32 range. Numbers 548 and above are
430 # not special and are not to be used for x32-specific syscalls.
431 548 common capture_memory_snapshot sys_capture_memory_snapshot
432 549 common track_syscall_usage sys_track_syscall_usage
```

2. Crear estructura accesible en `/include/linux/myTrackSyscall.h` así como los contadores usando `atomic` que servirá para los contadores; esto puede garantizar seguridad de concurrencia sin usar un bloqueo explícito como `mutex`. `Extern` indica que otros archivos tienen acceso a estos sin duplicar instancias. La definición debe ir en otro archivo `.c` y solo una vez.

```
1 #ifndef MY_TRACK_SYSCALL_H
2 #define MY_TRACK_SYSCALL_H
3 #include <linux/atomic.h>
4
5 // Declaraciones globales de contadores
6 extern atomic_t syscall_open_count;
7 extern atomic_t syscall_write_count;
8 extern atomic_t syscall_read_count;
9 extern atomic_t syscall_fork_count;
10
11 struct syscall_usage_stats {
12     unsigned long open_count; // Contador para llamadas a open()
13     unsigned long write_count; // Contador para llamadas a write()
14     unsigned long read_count; // Contador para llamadas a read()
15     unsigned long fork_count; // Contador para llamadas a fork()
16 };
17
18 #endif // MY_TRACK_SYSCALL_H
19
```

3. IMPLEMENTACION DE SYSCALL (/kernel/sys.c)

Importar la estructura a utilizar

```
#include <linux/mymemorySys.h> //Para encabezado de
#include <linux/myTrackSyscall.h> //Para encabezado
#include <linux/mmzone.h> // Para funciones de memor
```

Código: Recibe un struct del formato creado y solamente leeré el valor de los contadores para llenar con la información. De igual manera antes de la definición se definen los contadores inicializados en cero.

```
1 // CONTADOR DE LLAMADAS
2
3 // Definición de los contadores globales
4 atomic_t syscall_open_count = ATOMIC_INIT(0);
5 atomic_t syscall_write_count = ATOMIC_INIT(0);
6 atomic_t syscall_read_count = ATOMIC_INIT(0);
7 atomic_t syscall_fork_count = ATOMIC_INIT(0);
8
9 SYSCALL_DEFINE1(track syscall usage, struct syscall_usage_stats __user *, user_stats) {
10     struct syscall_usage_stats stats;
11
12     // Leer los valores de los contadores globales
13     stats.open_count = atomic_read(&syscall_open_count);
14     stats.write_count = atomic_read(&syscall_write_count);
15     stats.read_count = atomic_read(&syscall_read_count);
16     stats.fork_count = atomic_read(&syscall_fork_count);
17
18     // Copiar la estructura al espacio de usuario
19     if (copy_to_user(user_stats, &stats, sizeof(stats))) {
20         return -EFAULT;
21     }
22
23     return 0; // Exit
24 }
```

MODIFICACIONES:

1. READ: Al buscar en la tabla de llamadas vemos que tiene el nombre `sys_read`.

```
# The abi is "common", "64" or "x32" for this
#
0 common read sys_read
```

Su definición está en `/fs/read_write.c`, en este archivo se importa el archivo `.h` y se aumenta el respectivo contador:

```
627     return ret;
628 }
629
630 SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
631 {
632     // Incrementar el contador para la syscall 'read'
633     atomic_inc(&syscall_read_count);
634
635     return ksys_read(fd, buf, count);
636 }
637
638 ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
639 {
640     return ksys_write(fd, buf, count);
641 }
```

2. WRITE: Esta también se encuentra el archivo de la llamada anterior, por lo que el procedimiento solo aumenta el contador ya que la importación ya fue hecha.

```
1 common write sys_write
```

```
}  
  
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,  
| size_t, count)  
|  
| // Incrementar el contador para 'write'  
| atomic_inc(&syscall_write_count);  
|  
| return ksys_write(fd, buf, count);  
|
```

3. OPEN: Ser verifica la tabla de llamadas y su nombre se define en el siguiente directorio: fs/open.c

```
2 common open sys_open
```

En este archivo se importa el archivo .h y se aumenta el respectivo contador:

```
5 SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)  
|  
| // Incrementar el contador para open  
| atomic_inc(&syscall_open_count);  
|  
| if (force_o_largefile())  
| | flags |= O_LARGEFILE;  
| return do_sys_open(AT_FDCWD, filename, flags, mode);  
|
```

4. FORK:
Ser verifica la tabla de llamadas y su nombre se define en el siguiente directorio:
kernel/fork.c

```
57 common fork sys_fork
```

En este archivo se importa el archivo .h y se aumenta el respectivo contador:

POSIBLES ERRORES:

De manera común se puede pensar que se usa esta macro por su nombre, pero en realidad si usamos el comando strace ./test (donde test es un programa que usa fork()) veremos la siguiente salida:

```

5
6 #ifdef __ARCH_WANT_SYS_FORK
7 SYSCALL_DEFINE0(fork)
8 {
9     // Incrementar el contador para la syscall
10    atomic_inc(&syscall_fork_count);
11
12 #ifdef CONFIG_MMU
13    struct kernel_clone_args args = {
14        .exit_signal = SIGCHLD,
15    };
16
17    return kernel_clone(&args);
18 #else

```

```

prlimit64(0, RLIMIT_STACK, NULL, {rlim_
munmap(0x7c6a6acbd000, 75067)
clone(child_stack=NULL, flags=CLONE_CHI
087
Soy el proceso hijo, esperando 20 segun
fstat(1, {st mode=S_IFCHR|0620, st rdev
getrandom("\xcf\x77\x67\x9e\x42\x2e\xab

```

Vemos que no usa el macro de fork si no que utiliza “clone” por lo que en este caso la implementación es en esta parte:

```

22 SYSCALL_DEFINE3(clone, unsigned long, newsp, unsigned long, clone_flags,
23 #ifdef CONFIG_CLONE_BACKWARDS3
24 SYSCALL_DEFINE3(clone, unsigned long, clone_flags, unsigned long, newsp,
25 int, stack_size,
26 int __user *, parent_tidptr,
27 int __user *, child_tidptr,
28 unsigned long, tls)
29 #else
30 SYSCALL_DEFINE3(clone, unsigned long, clone_flags, unsigned long, newsp,
31 int __user *, parent_tidptr,
32 int __user *, child_tidptr,
33 unsigned long, tls)
34 #endif
35 #endif
36 {
37    struct kernel_clone_args args = {
38        .flags = (lower_32_bits(clone_flags) & ~CSIGNAL),
39        .pidfd = parent_tidptr,
40        .child_tid = child_tidptr,
41        .parent_tid = parent_tidptr,
42        .exit_signal = (lower_32_bits(clone_flags) & CSIGNAL),
43        .stack = newsp,
44        .tls = tls,
45    };
46
47    // Incrementar el contador para la syscall
48    atomic_inc(&syscall_fork_count);
49
50    return kernel_clone(&args);
51 #endif

```

4. Agregue también el prototipo en la ruta y archivo indicado:

```

asmlinkage long sys_track_syscall_usage(struct syscall_usage_stats __user *user_stats);

```

5. TESTEO:

```

Fork count: 1126
moises@moises-VirtualBox:~/Escritorio/test$ ./test2
Después de las acciones:
Open count: 269651
Write count: 62742
Read count: 155419
Fork count: 1129
moises@moises-VirtualBox:~/Escritorio/test$

```

get_io_throttle

Flujo:

0. Escoger un nombre get_io_throttle
1. Editar el archivo con la tabla del sistema para agregar la nueva llamada.

```

32 545 common track_syscall_usage sys_track_syscall_usage
33 550 common get io throttle sys get io throttle

```

2. Crear estructura accesible en /include/linux/myIOStatsSyscall.h para la información.

```

# syscall_h408b | C syscall.h | C syscall | C myIOStatsSyscall.h
1 #ifndef LINUX_IO_STATS_H
2 #define LINUX_IO_STATS_H
3
4 struct io_stats {
5     unsigned long bytes_read; // Bytes leídos
6     unsigned long bytes_written; // Bytes escritos
7     unsigned long bytes_cancelled; // Bytes cancelados
8     unsigned long maj_flt; // Fallos de página mayores
9     unsigned long min_flt; // Fallos de página menores
10 };
11
12 #endif /* LINUX_IO_STATS_H */
13

```


3. IMPLEMENTACION DE SYSCALL (/kernel/sys.c)

Importar la estructura a utilizar

```
85 #include <linux/vmstat.h> //Estado de memoria virtual
86 #include <linux/myIOStatsSyscall.h> //Estadísticas de un PID
87 //-----
```

Código: Recibe un struct del formato creado y el PID que se desea analizar y consultara con las estructuras definidas y explicadas a continuación para llenar de información y retornar.

```
33 // ESTADÍSTICAS DE UN PID
34 SYSCALL_DEFINE2(get_io_throttle, pid_t, pid, struct my_io_stats __user *, io_stats)
35 {
36     struct task_struct *task;
37     struct my_io_stats stats;
38
39     // Buscar el proceso por el pid
40     task = find_task_by_vpid(pid);
41     if (!task) {
42         return -ESRCH; //ERROR
43     }
44
45     // Obtener la información estadística de I/O del proceso
46     stats.bytes_read = task->iocb.read_bytes;
47     stats.bytes_written = task->iocb.write_bytes;
48     stats.bytes_cancelled = task->iocb.cancelled_write_bytes;
49     stats.maj_flt = task->maj_flt; // Fallos de página mayores
50     stats.min_flt = task->min_flt; // Fallos de página menores
51
52     if (copy_to_user(io_stats, &stats, sizeof(struct my_io_stats))) {
53         return -EFAULT; // Error
54     }
55
56     return 0;
57 }
```

```
1 //
2
3 struct task_struct {
4 #ifdef CONFIG_THREAD_INFO_IN_TASK
5     /*
6      * For reasons of header soup (see current_thread_info()), this
7      * must be the first element of task_struct.
8      */
9     struct thread_info thread_info;
10 #endif
11     unsigned int _state;
12
13     /* saved state for "spinlock sleepers" */
14     unsigned int saved_state;
15
16     /*
17      * This begins the randomizable portion of task_struct. Only
18      * scheduling-critical items should be added above here.
19      */
20     randomized_struct_fields_start
21
22     void *stack;
23     refcount_t usage;
24     /* Per task flags (PF_*), defined further below: */
25     unsigned int flags;
26     unsigned int ptrace;
27
28 #ifdef CONFIG_SMP
29     int on_cpu;
30     struct _call_single_node wake_entry;
31     unsigned int wakee_flips;
32     unsigned long wakee_flip_decay_ts;
33     struct task_struct *last_wakee;
34 }
```

Los fallos de página (page faults) en un sistema operativo están directamente relacionados con la gestión de la memoria virtual, y aunque no son una métrica exclusivamente de I/O (entrada/salida), están íntimamente vinculados a operaciones de I/O, especialmente cuando se trata de acceso a datos en disco o a memoria que no está actualmente en la RAM.

1. Fallos de página mayores (maj_flt):

Ocurre cuando un proceso intenta acceder a una página de memoria que no está presente en la memoria RAM y que debe ser cargada desde el disco (generalmente del swap o del almacenamiento físico).

Relación con I/O:

Los fallos de página mayores indican que el sistema ha tenido que realizar operaciones de I/O, lo cual es directamente relevante cuando se está midiendo la actividad de un proceso. Si un proceso tiene muchos fallos de página mayores, es probable que esté experimentando cuellos de botella relacionados con I/O, como el acceso frecuente al disco debido a la falta de memoria en RAM.

2. Fallos de página menores (min_flt):

Ocurre cuando un proceso intenta acceder a una página de memoria que no está en la RAM, pero esa página ya está en el sistema de archivos o en la memoria caché, y no es necesario realizar una operación de I/O para traerla del disco.

Relación con I/O:

Aunque no siempre impliquen I/O directo, los fallos de página menores son indicativos de cómo el sistema está manejando la memoria, y una alta frecuencia de estos podría sugerir una presión sobre los recursos de memoria, lo cual podría tener implicaciones en el uso general de I/O, particularmente en sistemas con memoria virtual intensiva.

```
struct task_io_accounting {
#ifdef CONFIG_TASK_XACCT
    /* bytes read */
    u64 rchar;
    /* bytes written */
    u64 wchar;
    /* # of read syscalls */
    u64 syscr;
    /* # of write syscalls */
    u64 syscw;
#endif /* CONFIG_TASK_XACCT */

#ifdef CONFIG_TASK_IO_ACCOUNTING
    /*
     * The number of bytes which this task has caused to be read from
     * storage.
     */
    u64 read_bytes;

    /*
     * The number of bytes which this task has caused, or shall cause,
     * to be written to disk.
     */
    u64 write_bytes;

    /*
     * A task can cause "negative" IO too. If this task truncates its
     * disk space, some IO which another task has been accounting

```

read_bytes: Es la cantidad de bytes que este proceso ha causado que se lean desde el almacenamiento (disk).

write_bytes: Es la cantidad de bytes que este proceso ha causado que se escriban en el almacenamiento.

cancelled_write_bytes: Es la cantidad de bytes que, aunque originalmente estaban destinados a ser escritos, fueron "cancelados" debido a un truncamiento de caché de páginas sucias (lo que puede suceder en ciertos contextos como la liberación de memoria o la escritura cancelada).

4. Agregue también el prototipo en la ruta y archivo indicado:

```
asmlinkage long sys_track_syscall_usage(struct syscall_usage_stats __user *user_stats);
asmlinkage long sys_get_io_throttle(pid_t pid, struct my_io_stats __user *io_stats);
// ++++++ FIN ++++++
```

5. TESTEO:

```
noises@moises-VirtualBox:~/Escritorio/test$ gcc -o test3 test3.c
noises@moises-VirtualBox:~/Escritorio/test$ ./test3

Estadísticas de I/O para el PID 1:
Bytes leídos: 20619264
Bytes escritos: 0
Bytes cancelados (escritos): 0
Fallos de página mayores: 104
Fallos de página menores: 16384
noises@moises-VirtualBox:~/Escritorio/test$
```

MODULOS

Se crearon 3 módulos para capturar 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:

- **Estadísticas de CPU:** El módulo debe obtener y registrar el porcentaje de uso de CPU.
- **Estadísticas de memoria:** Debe mostrar el uso actual de memoria total y memoria disponible.
- **Estadísticas de almacenamiento:** Muestra el espacio total y el espacio libre en el disco, específico a una partición indicada.

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 Syscalls

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <stdlib.h>
#include <errno.h>
#include <time.h>

#define SYS_CAPTURE_MEMORY_SNAPSHOT 548
#define SYS_track_syscall_usage 549
#define SYS_get_io_throttle 550

// Estructura para capturar el snapshot de memoria
> struct my_memory_snapshot { int
};

// Estructura para las estadísticas de las syscalls
> struct syscall_usage_stats { int
};

// Estructura para las estadísticas de I/O
> struct my_io_stats { int
};

void test_memory_snapshot() {
    struct my_memory_snapshot snapshot;

    long result = syscall(SYS_CAPTURE_MEMORY_SNAPSHOT, &snapshot);

    if (result == 0) {
        printf("++ Memory Snapshot ++\n");
        printf("Cached Memory: %lu KB\n", snapshot.memory_cache);
    }
}
```

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

Se definen los números de llamadas asignadas. Se crea las estructuras creadas y definidas para cada llamada.

Se definen métodos para invocar la llamada, llenar los struct de información e imprimir en consola.

2. Modulo CPU

```
// Estructura para almacenar el uso anterior de la CPU
static unsigned long prev_total_time = 0;
static unsigned long prev_idle_time = 0;

static int cpu_usage_show(struct seq_file *m, void *v) {
    unsigned long user_time, nice_time, system_time, idle_time, iowait_time, irq_time, softirq_time, total_time;
    unsigned long total_time_diff, idle_time_diff;
    unsigned long cpu_usage;

    // Abrir el archivo /proc/stat y leer la primera línea
    struct file *file = filp_open("/proc/stat", 0_RDONLY, 0);
    if (IS_ERR(file)) {
        seq_printf(m, "Error al abrir /proc/stat\n");
        return PIR_ERR(file);
    }

    // Leer la línea correspondiente al uso de la CPU
    char buf[256];
    loff_t pos = 0;
    ssize_t len = kernel_read(file, buf, sizeof(buf) - 1, &pos);
    filp_close(file, NULL);

    if (len <= 0) {
        seq_printf(m, "Error al leer /proc/stat\n");
        return -1;
    }
}
```

Se tiene como objetivo calcular el uso actual de la CPU en porcentaje y mostrarlo en formato JSON a través de un archivo en /proc.

Estas variables se utilizan para almacenar el total de tiempo y el tiempo de inactividad (idle) de la CPU en la última lectura. Son esenciales para calcular la diferencia en el tiempo transcurrido entre dos lecturas.

El módulo abre el archivo /proc/stat, que contiene estadísticas sobre el uso de la CPU. En este archivo, la primera línea (cpu) proporciona información sobre los tiempos acumulados en diferentes estados de la CPU:

user_time: Tiempo en modo usuario (sin nice).
nice_time: Tiempo en modo usuario con prioridad ajustada (nice).
system_time: Tiempo en modo kernel.
idle_time: Tiempo inactivo.
iowait_time: Tiempo esperando operaciones de E/S.
irq_time: Tiempo gestionando interrupciones de hardware.
softirq_time: Tiempo gestionando interrupciones de software.

```
// Asegurarse de que se leyó correctamente la línea
buf[len] = '\0';

// Procesar la primera línea de /proc/stat
if (sscanf(buf, "cpu %lu %lu %lu %lu %lu %lu %lu",
           &user_time, &nice_time, &system_time, &idle_time,
           &iowait_time, &irq_time, &softirq_time) != 7) {
    seq_printf(m, "Error al analizar la línea de CPU\n");
    return -1;
}

// Calcular el total de tiempo y la diferencia desde la última lectura
total_time = user_time + nice_time + system_time + idle_time + iowait_time + irq_time + softirq_time;
total_time_diff = total_time - prev_total_time;
idle_time_diff = idle_time - prev_idle_time;

// Calcular el porcentaje de uso de la CPU
cpu_usage = 100 * (total_time_diff - idle_time_diff) / total_time_diff;

// Actualizar los tiempos previos para la siguiente iteración
prev_total_time = total_time;
prev_idle_time = idle_time;

// Imprimir el resultado en formato JSON
seq_printf(m, "{\n");
seq_printf(m, "  \"cpu_usage\": \"%lu%%\"\n", cpu_usage);
seq_printf(m, "}\n");

return 0;
}
```

El módulo extrae los valores de la primera línea del archivo para los distintos estados de la CPU. Estos valores están en unidades de "ticks" desde el inicio del sistema.

El cálculo considera solo el tiempo ocupado (total_time_diff - idle_time_diff) como porcentaje del tiempo total transcurrido. Se actualizan las variables prev_total_time y prev_idle_time para que puedan ser usadas en la siguiente llamada.

3. Modulo DISCO

```
static int sysinfo_disk_show(struct seq_file *m, void *v) {
    struct kstatfs stat;
    struct file *file;
    unsigned long total, free, used;

    // Ruta del dispositivo montado
    const char *mount_path = "/";

    // Abrir el directorio raíz
    file = filp_open(mount_path, O_RDONLY, 0);
    if (IS_ERR(file)) {
        seq_printf(m, "Error: no se pudo abrir la ruta %s\n", mount_path);
        return PTR_ERR(file);
    }

    // Obtener información del sistema de archivos
    if (vfs_statfs(&file->f_path, &stat)) {
        seq_printf(m, "Error: no se pudo obtener la información del sistema de archivos\n");
        filp_close(file, NULL);
        return -EIO;
    }

    // Calcular tamaños en bytes
    total = stat.f_blocks * stat.f_bsize;
    free = stat.f_bfree * stat.f_bsize;
    used = total - free;

    // Convertir a MB para evitar punto flotante
    seq_printf(m, "Dispositivo montado: %s\n", mount_path);
    seq_printf(m, "Tamaño total: %lu MB\n", total / (1024 * 1024));
}
```

Este módulo toma la ruta del dispositivo montado y abre su directorio raíz.

Luego mediante `vfs_statfs` se obtienen las métricas del sistema de archivos en el cual se tiene el tamaño total y usado.

Luego se hace la conversión a MB.

4. Modulo Memoria

```
static int memory_stats_show(struct seq_file *m, void *v) {
    struct sysinfo si;
    si_meminfo(&si);

    unsigned long total_ram = si.totalram * (PAGE_SIZE / 1024);
    unsigned long free_ram = si.freeram * (PAGE_SIZE / 1024);
    unsigned long used_ram = total_ram - free_ram;

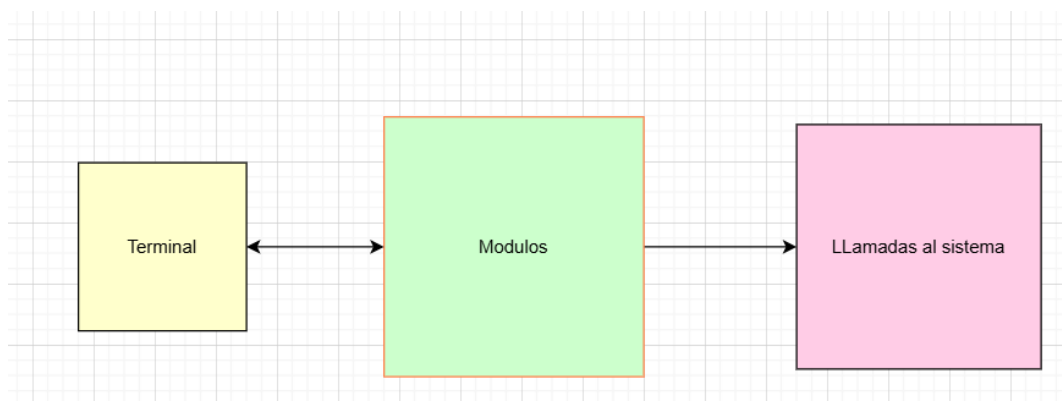
    seq_printf(m, "{\n");
    seq_printf(m, "  \"total_ramKB\": \"%lu\", \n", total_ram);
    seq_printf(m, "  \"free_ramKB\": \"%lu\", \n", free_ram);
    seq_printf(m, "  \"used_ramKB\": \"%lu\", \n", used_ram);
    seq_printf(m, "}\n");

    return 0;
}
```

Este módulo utiliza la estructura `sysinfo` con métricas de la memoria RAM.

Solo guarda los datos y le escribe al archivo en `/proc`.

ESQUEMA



USO DEL SISTEMA

```
moises@moises-VirtualBox:~/Escritorio/REP0/Proyecto1$ ./scriptVerificar.sh
Ejecutando ./getSyscalls...
++ ++ Memory Snapshot ++ ++:
Cached Memory: 391546 KB
Active Pages: 210148 KB
Inactive Pages: 164248 KB
Dirty Memory: 46 KB
Writeback Memory: 0 KB
Swap Total: 524287 KB
Swap Free: 524287 KB

++ Conteo de llamadas del sistema ++
Open count: 539339
Write count: 274893
Read count: 827065
Fork count: 2207

+++ Estadísticas de I/O para el PID 691 +++
Bytes leídos: 1310720
Bytes escritos: 0
Bytes cancelados (escritos): 0
Fallos de página mayores: 14
Fallos de página menores: 869
Mostrando contenido de /proc/cpu_usage...
{
  "cpu_usage": "8%"
}
-----
Mostrando contenido de /proc/memory_stats...
{
  "total_ramKB": "7404644",
  "free_ramKB": "4256824",
  "used_ramKB": "3147820"
}
-----
Mostrando contenido de /proc/myDiskInfo_202010833...
Dispositivo montado: /
Tamaño total: 44074 MB
Espacio usado: 32895 MB
Espacio disponible: 11179 MB
-----
```

Ejecute el script en bash (.sh) para llamar a las syscalls personalizadas, así como la información recopilada por los módulos.

Para ver el nombre y el mensaje de bienvenida puede ejecutar los siguientes comandos:

uname -s y dmesg -> El mensaje estará al principio.

Repositorio: https://github.com/Hrafnyr/SO2_202010833_VD2024/tree/main/Proyecto1

CRONOGRAMA:

8/12/24 – 14/12/24

SUNDAY	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY
	Configuración de entorno	Syscall de memoria		Syscall de métricas I/O	Modulo 4	Entrega de proyecto
	Compilación del kernel		Syscall de contador de llamadas	Modulo 1	Documentación	
				Modulo 2		
				Modulo 3		

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.

Se logró un nivel medio de manejo de errores ya que se logró identificar y resolver algunos errores y problemas durante el desarrollo del proyecto. No obstante, aún falta aspectos de mejora tales como posible optimización del código y reubicación para poder implementar el módulo de llamadas del sistema personalizadas que fue lo que faltó.

Esta mejora implica un manejo avanzado de errores de tal manera de que se implementen estrategias más robustas para encontrar la solución.