

---

Fecha de Entrega: 18 de enero, 2023.

Descripción: en este laboratorio se realizarán ejercicios para reforzar los conceptos de llamadas de sistema e introducir de manera práctica el concepto de proceso. Estos ejercicios requerirán programación en C. Se explicarán algunas instrucciones y métodos de trabajo usados con C, pero se recomienda profundizar e investigar sobre el lenguaje puesto que lo seguiremos usando durante el curso. Deberá entregar todos los archivos de código que programe (con instrucciones de compilación y ejecución si es necesario), así como un documento con las respuestas a las preguntas planteadas en cada ejercicio.

Materiales: se requerirá el software VirtualBox (<https://www.virtualbox.org/>) y la máquina virtual "OSConcepts" provista en la sección Módulos -> Máquinas Virtuales, dentro de Canvas. Puede trabajar todo en la versión de 32 bits (Debian "Squeeze", es decir, la propuesta en Canvas). Alternativamente, puede trabajar los primeros dos ejercicios en cualquier sistema Linux donde se pueda instalar `strace`. El último ejercicio debe ser realizado en "Squeeze".

### **Ejercicio 1 (20 puntos)**

- a. Escriba un programa en C (usando los programas `cat`, `nano` o `gEdit`) que despliegue el mensaje "Hello World!" y el número de identificación del proceso en el que se ejecuta el programa. Un programa ejecutable en C, como en Java, requiere la presencia de un método `main`. Para este ejercicio cree un método llamado `main`, sin parámetros, cuyo valor de retorno sea `int`. Debe incluir los encabezados `<stdio.h>` y `<unistd.h>` usando la directiva `#include`. Dentro del método `main` incluya las siguientes instrucciones:

```
printf("Hello World!\n");  
printf("%d\n", (int)getpid()); return  
(0);
```

Su programa se deberá ver así:

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(){  
    printf("Hello World!\n");  
    printf("%d\n", (int)getpid());  
    return (0);  
}
```

Guarde su programa con extensión `.c`

### Explicación:

La directiva `#include` permite la importación de código. La información encerrada entre corchetes triangulares indica el nombre del encabezado a importar, el cual es buscado en ubicaciones predeterminadas (y editables) en la configuración del compilador. Es posible incluir código encerrando el nombre del encabezado entre comillas en lugar de corchetes triangulares, pero esto modifica las ubicaciones de búsqueda.

Los encabezados son archivos que convencionalmente usan la extensión `.h` y cuyo propósito es declarar componentes que se pueden usar en diferentes partes de un mismo programa. El principio es similar (no igual) al de una interfaz en Java. Al compilar, todos los archivos con extensión `.c` se convierten en unidades de compilación (también llamadas unidades de traducción), que a su vez son transformadas en código objeto (convencionalmente con la extensión `.o`). Los archivos con el código objeto son unidos en un mismo programa ejecutable por el *linker*, y el *linker* se encarga de buscar entre los diferentes archivos las definiciones para componentes cuyas declaraciones estén presentes (normalmente, por medio de `#include`) en las unidades de traducción. En este programa se incluyen los encabezados `stdio.h` y `unistd.h`, que proveen llamadas al sistema para entrada y salida; y manejo de procesos.

La función `main` tiene `int` como tipo de retorno ya que, cuando concluye un programa, el resultado es un número que indica terminación exitosa o algún error. No es necesario, sin embargo, el uso de `return(0)`, ya que al completarse la ejecución de `main` exitosamente se retorna 0 automáticamente.

La función `printf` recibe como parámetro un *string* que puede contener especificadores de formato, denotados por un símbolo `'%'`. Si dicho *string* contiene especificadores de formato entonces se deben incluir parámetros adicionales que reemplacen, en orden, los especificadores de formato por sus valores. Lo que va después del símbolo de porcentaje debe incluir un carácter de entre varios que determinan qué será lo que se despliegue (por ejemplo, *d* para un número entero o decimal, *f* para uno de punto flotante); y puede tener varios otros modificadores que detallan cómo se despliega lo que se despliegue.

Por último, `getpid` es una llamada de sistema que devuelve el número del proceso que hace dicha llamada. Su resultado es un valor de tipo `pid_t` cuyo propósito es exclusivamente representar identificadores de procesos. Por ello *casteamos* `getpid()` a `int`, con tal de poder desplegarlo con `printf`.

- b. Ahora escriba otro programa en C con los mismos `#includes` que use la llamada a sistema `fork` y que almacene su resultado en una variable de tipo `int` llamada `f`. Luego agregue una condición que verifique si `f` es igual a 0. Tanto en el *if* como en el *else*, ejecute la llamada a sistema

`execl` que sirve para ejecutar el programa dado en sus parámetros. El primer parámetro será un *string* con el nombre del ejecutable (ver inciso c) de nuestro primer programa y el segundo será `(char*) NULL`. En el *else*, antes de la ejecución de `execl`, despliegue nuevamente el número de proceso como lo hizo en el programa anterior. Guarde su programa con extensión `.c`.

- c. Finalmente compilaremos y ejecutaremos los programas. Para compilar, use la siguiente instrucción desde una terminal ubicada en el directorio donde están sus programas:

```
gcc -o nombredelejecutable nombredelprograma.c
```

- Compile el primer programa y ejecútelo varias veces. Responda: ¿por qué aparecen números diferentes cada vez?
  - **Aparecen diferentes números debido a que se crea un nuevo proceso con diferente ID cada vez que se ejecuta el programa. Esto es debido a la función `getpid()`.**
- Proceda a compilar el segundo programa y ejecútelo una vez. ¿Por qué aparecen dos números distintos a pesar de que estamos ejecutando un único programa?
  - **Aparecen dos números diferentes por el `fork` que tenemos en el código, el cual crea un segundo proceso con copia exacta del programa. Esto significa que entonces tenemos dos procesos abiertos.**
- ¿Por qué el primer y el segundo números son iguales?
  - **El primer y segundo números son iguales, ya que estos se ejecutan por medio del mismo proceso.**
- En la terminal, ejecute el comando `top` (que despliega el *top* de procesos en cuanto a consumo de CPU) y note cuál es el primer proceso en la lista (con identificador 1). ¿Para qué sirve este proceso?
  - **El proceso con el identificador 1 es `root`. Este es el proceso de inicialización de Linux, este también es el encargado de inicializar los otros servicios del sistema y de iniciar otros procesos. Generalmente, este procesos tiene el PID de 1.**

## Ejercicio 2 (30 puntos)

- Investigue acerca de las llamadas a sistema `open()`, `close()`, `read()` y `write()`.
- Escriba un programa en C que reciba dos argumentos desde terminal (usando `int argc`, `char* argv[]` como parámetros en `main`). Su programa debe abrir el archivo ubicado en el directorio provisto como primer parámetro, y debe copiar todo su contenido a otro archivo ubicado en el directorio provisto como segundo parámetro.
- Instale la herramienta `strace` usando el siguiente comando en la terminal:

```
sudo apt-get install strace
```

**Importante:** si está usando la máquina virtual de 32 bits del libro (Debian “Squeeze”) deberá añadir repositorios al manejador de paquetes para instalar `strace`. Para hacerlo, diríjase en terminal al directorio `/etc/apt` y cree una copia del archivo `sources.list` llamada `sources.list.old`, así:

```
sudo cp sources.list sources.list.old
```

Luego abra `sources.list` en algún editor de texto (con `sudo`) y añada las siguientes líneas:

```
deb http://archive.debian.org/debian squeeze main deb  
http://archive.debian.org/debian squeeze-lts main
```

Al terminar, grabe sus cambios y cierre `sources.list`. Luego cree un archivo llamado `apt.conf` en el mismo directorio, que contenga lo siguiente:

```
Acquire::Check-Valid-Until false;
```

Finalmente actualice los repositorios de `apt-get` con la siguiente instrucción:

```
sudo apt-get update
```

Cuando esto concluya, intente instalar `strace` nuevamente.

- d. Compile su programa y pruébelo (no olvide crear un archivo qué copiar). Luego, ejecute el siguiente comando en terminal (desde la ubicación donde esté el código objeto resultante de la compilación):

```
strace ./<nombre del archivo con código objeto> <nombre de  
archivo fuente> <nombre de archivo destino>
```

- Observe el resultado desplegado. ¿Por qué la primera llamada que aparece es `execve`?
  - La primera llamada que aparece es `execve`, ya que esta es la primera llamada del sistema que se realiza cuando se ejecuta el programa. Esta también es la primera llamada, ya que esta también se utiliza para ejecutar una nueva instancia del programa.
- Ubique las llamadas de sistema realizadas por usted. ¿Qué significan los resultados (números que están luego del signo '=')?
  - Los números que están después de las llamadas de sistemas realizadas por mi mismo significan el valor de retorno de la llamada al sistema. El valor será positivo, si la llamada fue ejecutada con éxito y el numero significa el numero de proceso. Pero si el valor del numero es negativo, esto significa que hubo un error en la llamada y el numero puede ser un código de error.
- ¿Por qué entre las llamadas realizadas por usted hay un `read` vacío?
  - En mi caso la llamada dice `read(3, "", 4096) = 0`. Esto significa que no se lograron leer los bytes, que en esta caso eran 4096 bytes del descriptor del archivo 3.
- Identifique tres servicios distintos provistos por el sistema operativo en este `strace`. Liste y explique brevemente las llamadas a sistema que corresponden a los servicios identificados (puede incluir `read`, `write`, `open` o `close` que el sistema haga por usted, no los que usted haya producido directamente con su programa).
  - `brk`: La llamada de sistema `brk()` es utilizada para poder reservar memoria dinámicamente en tiempo de ejecución. Esto significa que el proceso puede aumentar o reducir su área de memoria según sea necesario. El `os` utiliza esta llamada para gestionar la memoria del proceso y garantizar que este tenga acceso a la memoria que necesita para su ejecución.
  - `mprotect`: Esta llamada de sistema se utiliza para poder cambiar los permisos de acceso a un área específica de memoria asignada al proceso en tiempo de la ejecución. Los permisos de esta llamada pueden ser para lectura, escritura o ejecución. Esta llamada

**también protege áreas de memoria para que no puedan ser modificadas o ejecutadas accidentalmente.**

- **mmap:** Esta llamada es utilizada para poder mapear un archivo o dispositivo en la memoria del sistema, lo cual significa que el contenido de un archivo puede ser accedido y modificado directamente como si fuera porción de la memoria del proceso.

### **Ejercicio 3 (50 puntos)**

En este ejercicio agregaremos una llamada de sistema al sistema operativo. Los pasos que se realizarán en este ejercicio pueden tomar bastante tiempo y tienen el potencial de arruinar nuestra máquina virtual. Se recomienda crear un *snapshot* de la máquina antes de continuar y antes de los pasos que tomen más tiempo. **Recordatorio:** use la versión de 32 bits de la máquina virtual.

- a. Copie el código fuente del sistema operativo al directorio *home* del usuario *os* con los siguientes comandos en una terminal:

```
cd ~ & sudo cp -a /usr/src/linux-  
2.6.39.4 .
```

- b. Agregue la referencia a su nueva llamada de sistema. Primero debe abrir la tabla de llamadas de sistema con los siguientes comandos:

```
cd linux-2.6.39.4/arch/x86/kernel/  
sudo nano syscall_table_32.S
```

Luego agregará su llamada de sistema a esta lista. Agregue al final del archivo lo siguiente:

```
.long sys_mycall          /* 345 */
```

Grabe sus cambios y salga.

- c. Póngale nombre al número con el que se accede a su llamada de sistema. Primero debe abrir el encabezado donde se definen las referencias numéricas para estas llamadas:

```
cd ../include/asm  
sudo nano unistd_32.h
```

Luego agregará la referencia numérica. Agregue (casi al final del archivo) luego de la línea  
#define \_\_NR\_syncfs 344, lo siguiente:

```
#define __NR_mycall 345
```

Finalmente modifique el número de llamadas de sistema para que refleje la nueva cantidad. La línea `#define NR_syscalls 345` deberá decir ahora:

```
#define NR_syscalls 346
```

Grabe sus cambios y salga.

- d. Agregue la declaración de su llamada de sistema. Primero deberá abrir el archivo de declaraciones de llamadas a sistema con los siguientes comandos:

```
cd ~/linux-2.6.39.4/include/linux sudo  
nano syscalls.h
```

Luego agregue al final del archivo, antes del `#endif`, la siguiente línea:

```
asmlinkage long sys_mycall(int i);
```

Grabe sus cambios y salga.

- e. Agregue un directorio para la implementación de su llamada a sistema y diríjase a él. Use las siguientes instrucciones:

```
sudo mkdir ~/linux-2.6.39.4/mycall  
cd ~/linux-2.6.39.4/mycall
```

Cree un programa en C que tenga la definición de su función `sys_mycall`. Este programa debe ser guardado con el nombre `mycall.c` y debe tener la directiva `#include <linux/linkage.h>`. Su función debe estar definida de la siguiente manera:

```
asmlinkage long sys_mycall(int i){  
    return i+10;  
}
```

Reemplace 10 por un número de su preferencia. En el mismo directorio creará un *Makefile* para compilar su definición. Cree un nuevo archivo, llamado *Makefile*, y agregue a él la siguiente línea:

```
obj-y := mycall.o
```

Grabe sus cambios y salga.

- f. Agregue su directorio al *Makefile* con el que se compilará el *kernel*. Primero debe abrir dicho *Makefile* usando las siguientes instrucciones:

```
cd ~/linux-2.6.39.4
```

```
sudo nano Makefile
```

Luego presione `Ctrl+W` para buscar texto e ingrese `core-y`. Repita este paso para encontrar la siguiente línea:

```
core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

Agregue, al final de esta línea, “`mycall/`” (nótese el espacio precedente). Grabe sus cambios y salga.

- g. Ahora debe recompilar el *kernel* e instalarlo. Para ello primero debe crear el archivo de configuración que detallará las características que deberá poseer el *kernel* compilado. Primero ejecute los siguientes comandos:

```
sudo make clean  
sudo make menuconfig
```

Esto abrirá un menú con muchas opciones, todas preconfiguradas a algún valor. Presione `Exit`, y cuando se le pregunte si desea grabar su configuración responda `Yes`. Al concluir esto deberá compilar el *kernel* con el siguiente comando:

```
sudo make
```

Esto tomará un buen tiempo en completarse. Vaya a tomarse un su café; chequee cada quince minutos. Cuando haya terminado será necesario compilar los módulos del *kernel*. Para ello ejecute el siguiente comando:

```
sudo make modules
```

Esto también tomará un tiempo. Cuando haya terminado procederemos a la instalación. Ejecute los siguientes comandos:

```
sudo make modules_install sudo  
make install
```

El primero instalará los módulos copiando los archivos necesarios al directorio `/lib/2.6.39.4`. El segundo instalará el nuevo *kernel* creando un archivo en `/boot` llamado `vmlinuz-2.6.39.4`.

Habiendo concluido estos pasos es necesario crear un sistema de archivos para que el nuevo *kernel* pueda iniciar (más acerca de esto a lo largo del curso). Esto lo realizará con el siguiente comando:



```
sudo update-initramfs -c -k 2.6.39.4
```

La opción `-c` provoca la creación del sistema de archivos y la opción `-k` especifica la versión del *kernel* para la cual se crea. Esto deberá agregar el archivo `initrd.img-2.6.39.4` al directorio `/boot`. Finalmente actualizará GRUB. GRUB es el programa que se encarga de cargar el sistema operativo (*bootstrap loader*), y su actualización detectará la existencia de un nuevo *kernel* que podremos elegir al iniciar la máquina virtual. Para ello ejecute el siguiente comando:

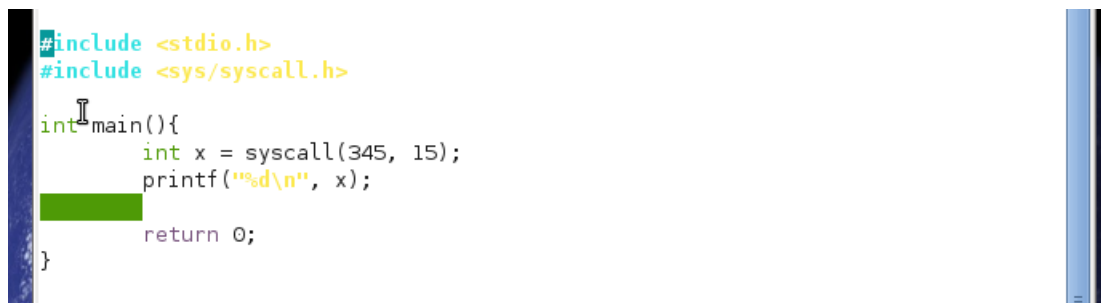
```
sudo update-grub
```

Luego de esto reinicie el sistema (la máquina virtual).

- h. Si todo salió bien, al iniciar deberá ver un menú con cuatro opciones, una de las cuales es nuestro *kernel* versión 2.6.39.4. Elija esa opción pulsando *Enter*. El sistema operativo deberá iniciar como si nada; proceda a abrir una terminal. Ahora creará un programa en C para probar su llamada a sistema. Su programa debe incluir los encabezados `stdio.h` y `sys/syscall.h`, y tendrá un `main` sin parámetros con las siguientes instrucciones:

```
int x = syscall(345, 15);  
printf("%d\n", x);
```

Compile y ejecute su programa. Si se despliega la suma de 15 con el número que usted haya dejado en la llamada a sistema, ha agregado exitosamente su propia llamada de sistema. **Nota:** puede revisar que el sistema operativo cuente con la llamada `sys_mycall` usando el siguiente comando en la terminal:



```
#include <stdio.h>  
#include <sys/syscall.h>  
  
int main(){  
    int x = syscall(345, 15);  
    printf("%d\n", x);  
    return 0;  
}
```



```
os@debian:~/Documents$ ./ej3  
115  
os@debian:~/Documents$
```

```
cat /proc/kallsyms | grep mycall  
os@debian:~/Documents$ cat /proc/kallsyms | grep mycall  
c113a818 T sys_mycall  
os@debian:~/Documents$ S
```

También puede verificar que su código esté siendo ejecutado por el *kernel* realizando un `strace` sobre su programa de prueba.

- ¿Qué ha modificado aquí, la interfaz de llamadas de sistema o el API? Justifique su respuesta.
  - **En este caso se modifica la interfaz de llamadas de sistema. Esto es debido a que a la hora de crear y llamar mi propio encabezado <sys/syscall.h> estoy usando una interfaz diferente y no estoy usando su API. Esto hace que pueda utilizar una interfaz diferente de la que me proporciona C y también permito que mi código logre acceder a servicios del kernel de mi sistema.**
- ¿Por qué usamos el número de nuestra llamada de sistema en lugar de su nombre?
  - **Estamos usando un numero entero en vez de su nombre, debido a que la función `syscall()` toma un numero entero como parámetro. Este entero especifica el numero de la llamada del sistema, y debido a que cada llamada al sistema tiene su entero único asociado, logramos llamar a la llamada correcta.**
- ¿Por qué las llamadas de sistema existentes como `read` o `fork` se pueden llamar por nombre?
  - **A diferencia de las sistema calls que son de bajo nivel, las llamadas de sistema existentes como `read` y `fork` son llamadas de alto nivel y están metidas dentro de una librería. Debido al hecho que estas llamadas existen dentro de la biblioteca de C, la cual nos proporciona una interfaz para poder llamar dichas funciones. Estas funciones las invocamos a través del encabezado <unistd.h>, y con esto ya podemos llamar a las funciones por su nombre, como `read()` y `fork()`.**

