Universidad de los Andes	Departamento de Ingeniería de Sistemas y Computación ISIS1311 – Tecnología e Infraestructura de Cómputo (TIC) **LABORATORIO 3.** **SISTEMA OPERATIVO, GESTIÓN DE PROCESOS**

CONTEXTO

Los programas desarrollados en un lenguaje de alto nivel pasan por un proceso de compilación que genera un archivo binario. El archivo binario o ejecutable es organizado por el compilador en secciones que pueden ser fácilmente identificadas por el sistema operativo, el cual se encarga, al momento de la ejecución del programa, de cargar estas secciones en ciertas ubicaciones de memoria para luego iniciar la ejecución del respectivo programa. Un programa en ejecución es conocido como proceso. Nótese que el programa por sí mismo es una entidad pasiva; sin embargo, el proceso es una entidad activa porque tiene estado. Su estado cambia a medida que la ejecución del programa avanza. Nótese, de igual forma, que el programa solo es una pequeña parte del proceso.

En la **Figura 1** se muestra, en la parte izquierda, el mapa de memoria del proceso que se describe en la parte derecha. Nótese que el proceso tiene unas estructuras de datos adicionales al programa que permiten mantener su estado. Estas estructuras, en conjunto, se conocen como **Process Control Block (PCB)** o Bloque de Control de Procesos.

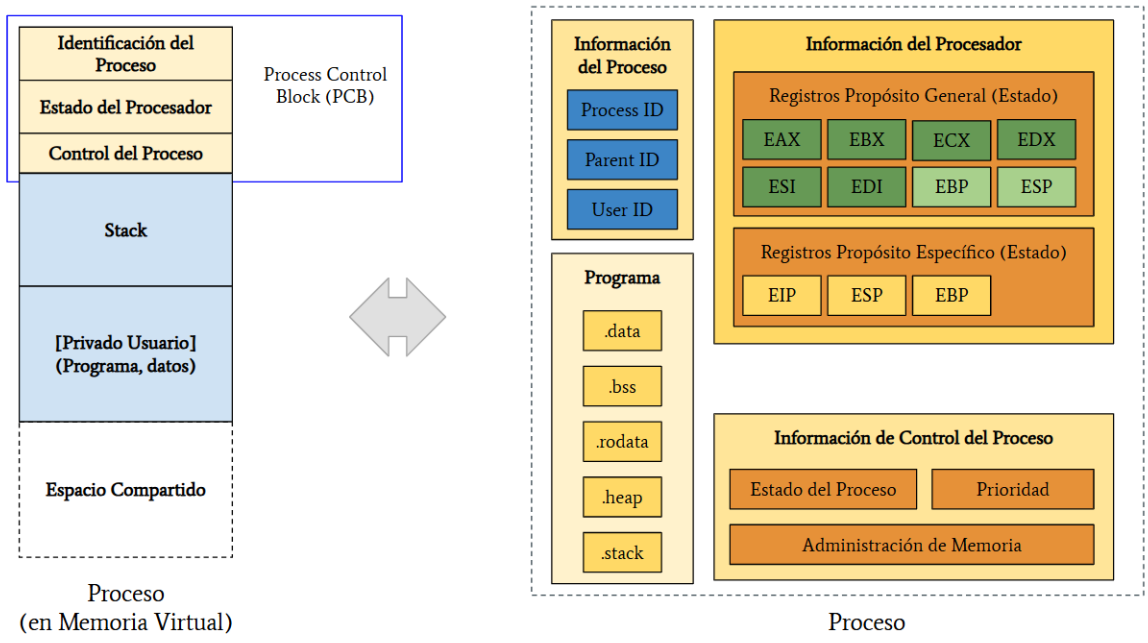

Figura 1. Proceso y Mapa de Memoria del Proceso

Descripción del Proceso de Ejecución de un Programa en Linux

Paso 1. Invocación del programa

Cuando un usuario ejecuta un programa (por ejemplo ***./mi_programa***) desde la terminal:

1. El terminal encuentra el archivo ***mi_programa*** en el directorio actual.
2. El terminal realiza el llamado al sistema ***fork()*** → crea un proceso hijo.
3. El hijo hace el llamado al sistema ***execve()*** → carga ***mi_programa*** en memoria y lo ejecuta.
4. El padre (la terminal) llama a ***wait()***.

Paso 2. Carga del programa por el Kernel

Cuando ***execve()*** es invocado:

1. El **kernel verifica permisos**, tipo de archivo (ELF en Linux), y validez del binario.
2. Se prepara el **espacio de direcciones del proceso** y se mapean las secciones del ejecutable en memoria virtual:

Secciones de memoria típicas en un ejecutable ELF:

Sección	**Contenido**	**Cómo se mapea**
.text	Código ejecutable	Solo lectura, ejecutable
.rodata	Datos constantes	Solo lectura
.data	Variables globales inicializadas	Lectura/escritura
.bss	Variables globales no inicializadas	Inicializado a 0, lectura/escritura
heap	Memoria dinámica (malloc)	Crece hacia direcciones altas de memoria con brk/mmap
stack	Variables locales y control de funciones	Crece hacia direcciones bajas de memoria, gestionado por el kernel
mmap	Librerías compartidas, archivos mapeados	Lectura/escritura o solo lectura según necesidad

3. El kernel **mapea dinámicamente las librerías compartidas** (libc.so, etc.) usando mmap().

Paso 3. Preparación del entorno de ejecución

Antes de entrar al código de usuario:

1. Se colocan en la **pila** (***stack***) los argumentos ***argv*** y ***envp***.
2. Se preparan **estructuras internas del kernel**, como:
    - task_struct (PCB del proceso)
    - mm_struct (descripción de memoria virtual)
3. El kernel establece el **registro de instrucción inicial (RIP/EIP)** apuntando a la función de entrada del programa.

Paso 4. Punto de entrada del programa

En Linux, la ejecución no empieza directamente en ***main()***:

1. El ejecutable ELF define un **entry point**, generalmente ***_start***.
2. _start realiza:
    - Inicialización del entorno C: ***stack***, ***heap***, variables globales.
    - Llamada a funciones de inicialización de librerías ***(.init_array***).
    - Finalmente, llama a ***main(argc, argv, envp)***.
3. Cuando ***main*** retorna, ***_start*** llama a ***exit()*** del sistema operativo.

Paso 5. Ejecución del programa

Durante la ejecución:

- **Funciones de C y llamadas a sistema**: cada función de biblioteca que interactúa con el SO termina llamando a un **system call** (***read()***, ***write()***, ***mmap()***, ***open()***, ***close()***, etc.).
- El CPU ejecuta instrucciones en la sección ***.text***.
- Variables globales usan ***.data*** o ***.bss***.
- Variables locales y parámetros de función usan ***stack***.
- Memoria dinámica usa ***heap***, gestionada por ***malloc***/***free*** (***brk()*** o ***mmap()*** internamente).

Paso 6. Finalización del programa

Cuando ***main()*** retorna:

1. ***_start*** llama a ***exit(status)***.
2. ***exit()***:
    - Cierra descriptores de archivos abiertos.
    - Libera memoria asignada al proceso.
    - Notifica al **proceso padre** mediante SIGCHLD.
    - Cambia el estado del proceso a ***zombie*** hasta que el padre hace ***wait()***.

Llamadas al Sistema Clave en Linux

Etapa	**Llamadas típicas**
Creación de proceso	*fork()*
Reemplazo de memoria	*execve()*
Mapeo de memoria	*mmap(), brk()*
IO	*read(), write(), open(), close()*
Finalización	*exit(), wait()*

ELF (Executable and Linkable Format)

- **ELF** es el formato estándar de archivos ejecutables en Linux y en la mayoría de sistemas tipo Unix.
- Es una **estructura** que organiza cómo está guardado un programa compilado en disco (antes de que se ejecute).

¿Qué contiene un archivo ELF?

Un archivo ELF no es solo "código". Incluye varias secciones que el sistema operativo necesita para cargarlo en memoria:

1. **Cabecera (Header):**
   Contiene información básica: tipo de archivo, arquitectura (32/64 bits), dirección de inicio del programa, etc.
2. **Secciones de código y datos:**
   - .text → instrucciones del programa (código).
   - .data → variables globales inicializadas.
   - .bss → variables globales sin inicializar.
   - .rodata → datos de solo lectura (constantes).
3. **Información para el enlazador y cargador:**
   - Dónde ubicar cada sección en memoria.
   - Qué librerías necesita el programa (por ejemplo libc.so).

¿Por qué es importante ELF?

- El **kernel de Linux entiende el formato ELF**.
- Cuando se ejecuta un programa (execve()), el kernel **lee la cabecera ELF** y usa esa información para:
  - Crear el espacio de direcciones del proceso.
  - Mapear cada sección en la memoria (código, datos, pila, heap).
  - Saber dónde empieza la ejecución del programa.

Ejemplo:

Si ejecutas en Linux:

```
file /bin/ls
```

El resultado será algo como:

```
/bin/ls: ELF 64-bit LSB executable, x86-64, dynamically linked, ...
```

Eso significa que /bin/ls es un binario en formato ELF, de 64 bits, listo para ser cargado por el kernel.

En resumen:

ELF es el "contenedor" que organiza el código, datos y metadatos de un programa para que el sistema operativo pueda cargarlo y ejecutarlo.

PREPATACIÓN PARA LAS ACTIVIDADES

1. Ingrese a Bloque Neon
2. Descargue el archivo ZIP que contiene el código fuente de las actividades.
3. Descomprima la carpeta
4. Ingrese a TICLAB https://ticlab.virtual.uniandes.edu.co/
5. Arrastre la carpeta al administrador de proyectos de Visual Studio Code en TICLAB.
6. Abra una terminal usando *Ctrl+ñ*

ACTIVIDAD 1. EXPLORANDO LAS SECCIONES DE UN PROGRAMA EN C

El objetivo de esta actividad es que el (la) estudiante reconozca las diferentes secciones de un ejecutable (.text, .data, .bss, .rodata) y comprenda cómo se organizan los datos del programa (código, variables locales, parámetros y sus valores).

PROCEDIMIENTO

1. Diríjase al directorio **1_Secciones_Programa**.
2. Abra el archivo ***program.c***
3. Identifique las variables globales (inicializadas y no inicializadas), variables locales y valores de solo lectura, por ejemplo: "Hello, sections!\n".
4. Use el siguiente comando para compilar el programa y generar un archivo .o (object)

```
clang -m32 -masm=intel -fms-extensions -O0 program.c -o program
```

5. Inspeccione las secciones del archivo binario usando ***objdump***. Únicamente copie el comando, no incluya los comentarios.

```
objdump -d -M intel program   # Desensamblar instrucciones (.text)
objdump -s -j .data program   # Ver contenido de .data
objdump -s -j .bss program    # Ver contenido de .bss
objdump -s -j .rodata program # Ver contenido de .rodata
```

Sección	Contenido	Ejemplo en el programa
.text	Código ejecutable (instrucciones)	Función main, llamadas a printf
.data	Variables globales inicializadas	int g_initialized = 42;
.bss	Variables globales no inicializadas (0 por defecto)	int g_uninitialized;
.rodata	Datos de solo lectura (constantes, cadenas)	"Hello, sections!\n"
Heap	Memoria dinámica asignada en ejecución	malloc(sizeof(int)) → heap_var
Stack	Variables locales, parámetros y direcciones	int local_var = 7;

CONCLUSION

- ***.text*** contiene el código ejecutable.
- ***.data*** guarda variables globales inicializadas.
- ***.bss*** almacena variables globales no inicializadas (se rellenan con ceros en ejecución).
- ***.rodata*** contiene constantes y literales de solo lectura.
- El ***stack*** se usa en tiempo de ejecución para variables locales y parámetros de funciones. La pila crece hacia direcciones más bajas de memoria. Se gestiona con ***push***, ***pop*** y **EBP** en ASM.
- El ***heap*** se usa en tiempo de ejecución para reservar espacio en memoria dinámica, es decir espacio reservado en tiempo de ejecución. El heap crece hacia direcciones más altas. Se gestiona con ***malloc*** y **free** en C.
- Con ***objdump*** podemos inspeccionar las secciones de un binario y confirmar la organización del programa en memoria.

ACTIVIDAD 2. DE PROGRAMA A PROCESO: EL CAMINIO HACIA LA EJECUCIÓN

El objetivo de esta actividad es que el (la) estudiante reconozca el proceso de ejecución de un programa, identificando el rol del sistema operativo, en particular Linux, en dicho proceso. Asimismo, se busca que observe la interacción del proceso con el sistema operativo a través de la **interfaz de llamadas al sistema**. La **Figura 2** ilustra un ejemplo de interacción directa entre el proceso y la interfaz de llamadas al sistema, mientras que la **Figura 3** muestra la interacción mediada por una biblioteca, que constituye el mecanismo más utilizado por las personas programadoras.

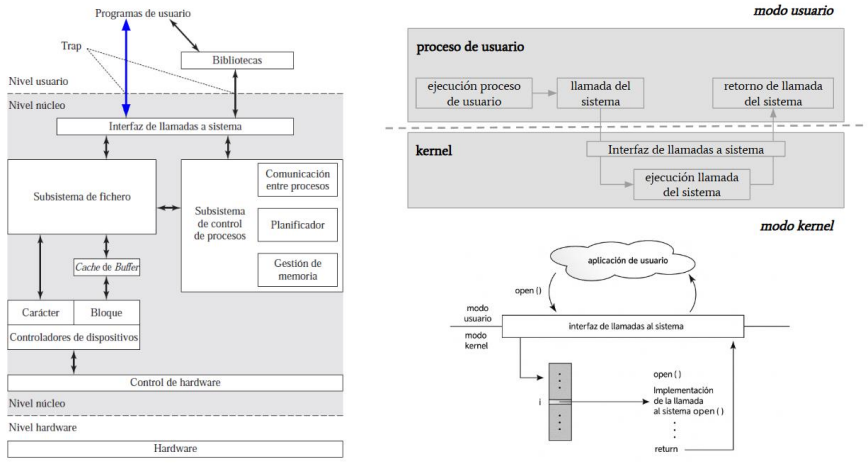

Figura 2. Proceso y Mapa de Memoria del Proceso[1]

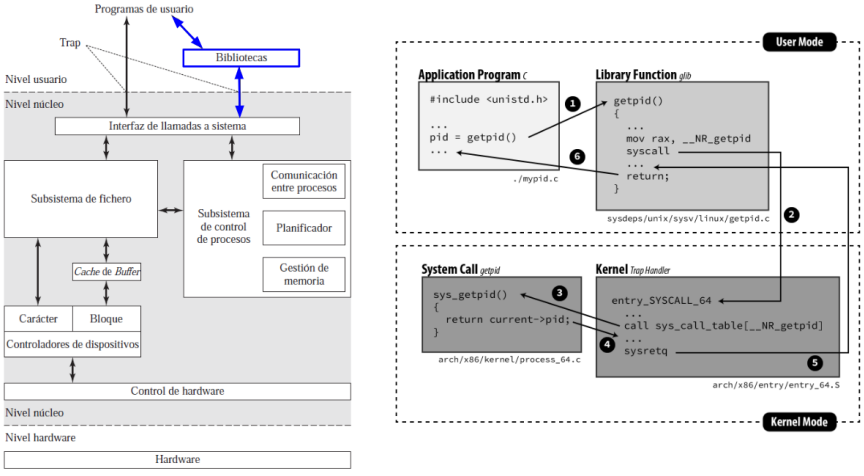

Figura 3. Proceso y Mapa de Memoria del Proceso[2]

[1] Referencias: Sistemas Operativos (5ed) - William Stallings, Página 93; Fundamentos de Sistemas Operativos (7ed) - Silberschatz

[2] Referencias: (1) Sistemas Operativos (5ed) - William Stallings, Página 93; (2) https://juliensobczak.com/inspect/2021/08/10/linux-system-calls-under-the-hood/

PROCEDIMIENTO (*INSPECCIÓN DE LLAMADOS AL SISTEM*)

1. Diríjase al directorio **2_De_Programa_a_Proceso**.
2. Abra el archivo ***program.c***
3. Inspeccione el programa en C brevemente para identificar lo que está haciendo.
4. Use el siguiente comando para compilar el programa

```
clang -m32 -masm=intel -fms-extensions -O0 program.c -o program
```

5. Observacioón con ***strace***

```
strace ./program
```

6. Identifique las llamadas al sistema. No es necesario que entienda el formato, únicamente debe identificarlas en la salida de ***strace***.

   Ejemplo de llamadas al sistema:

   - ***execve*** → creación del proceso.
   - ***brk***, ***mmap*** → asignación de memoria.
   - ***write*** → impresión en pantalla.
   - ***exit_group*** → terminación del proceso.

PROCEDIMIENTO (*INSPECCIÓN DEL PROCESS CONTROL BLOCK*)

1. Diríjase al directorio **3_Process_Control_Block**.
2. Abra el archivo ***program.c***
3. Inspeccione brevemente el programa en C para identificar lo que realiza. En particular, nótese que este programa es igual al anterior, pero incluye la función ***sleep(120)***. Esta función permite pausar la ejecución durante X segundos. Se utiliza para evitar que el programa termine de inmediato y así poder observar la información de su **Process Control Block (PCB).** Lo anterior es importante porque, una vez que el programa finaliza, dicha información es eliminada.
4. Use el siguiente comando para compilar el programa

```
clang -m32 -masm=intel -fms-extensions -O0 program.c -o program
```

5. Ejecutar le programa en segundo plano

```
./program &
[1] 1234   # PID = 1234
```

6. Note que al ejecutar el comando con &, el programa se ejecuta en segundo plano y retorna el **Process Identifier (PID)**. En su caso este número puede ser distinto y varía entre ejecuciones. Debe usar este comando en los siguientes para inspeccionar el PCB del proceso.
7. Consultar el estado del proceso, reemplace <PID> por el PID de su proceso.

```
cat /proc/<PID>/status
```

8. Identifique en la salida del comando anterior: “State:”, “PID:”, “PPID”, “VmPeak”, “VmSize”, “VmSwap”, “Threads”, “Cpus_allowed_list”, “voluntary_ctxt_switches”, “nonvoluntary_ctxt_switches”. Use ChatGPT para entender cuál es el rol y/o significado de estos atributos.
9. Consultar el mapa de memoria

```
cat /proc/<PID>/maps
```

10. Copie la salida del comando anterior en ChatGPT, pida que le índice cómo se interpreta esa salida.

CONCLUSION

- Un programa en disco se convierte en un proceso gestionado por el sistema operativo al ejecutarse.
- El proceso posee un PCB que contiene información esencial para su administración.
- La interacción con el ***kernel*** ocurre a través de ***llamadas al sistema*** y puede observarse con ***strace***.
- El sistema operativo juega un papel fundamental en la ejecución de programas: gestiona recursos, provee un entorno de ejecución seguro, organiza la memoria, administra procesos y garantiza la comunicación con el hardware. Sin el sistema operativo, los programas no podrían ejecutarse de manera controlada y eficiente.