Sistemas Operativos. Práctica: "Process Management System Calls"

Ricchy Alain Pérez Chevanier
Angel Renato Zamudio Malagón

1. Objetivo

El objetivo de esta práctica es que el alumno implemente las *system calls* de administración de procesos. Dicha implementación se realizará mediante interrupciones de hardware que provocan que un proceso de usuario cambie el modo de ejecución de usuario a kernel.

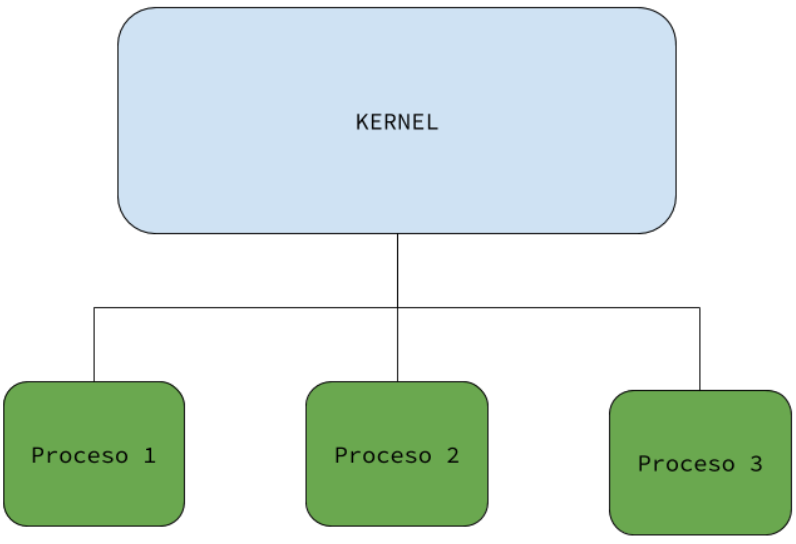
2. Introducción

2.1. Procesos de usuario

Un sistema operativo basado en procesos se construye con la finalidad de administrar procesos de usuario. En este sentido, los procesos de usuario deben consumir la mayor cantidad de recursos del sistema en comparación con el sistema operativo. La idea general de tener procesos de usuario se basa en el argumento de que algunos procesos deben tener acceso restringido a algunos recursos. Restringir el acceso a los recursos es un tema completamente de seguridad y estabilidad del sistema.

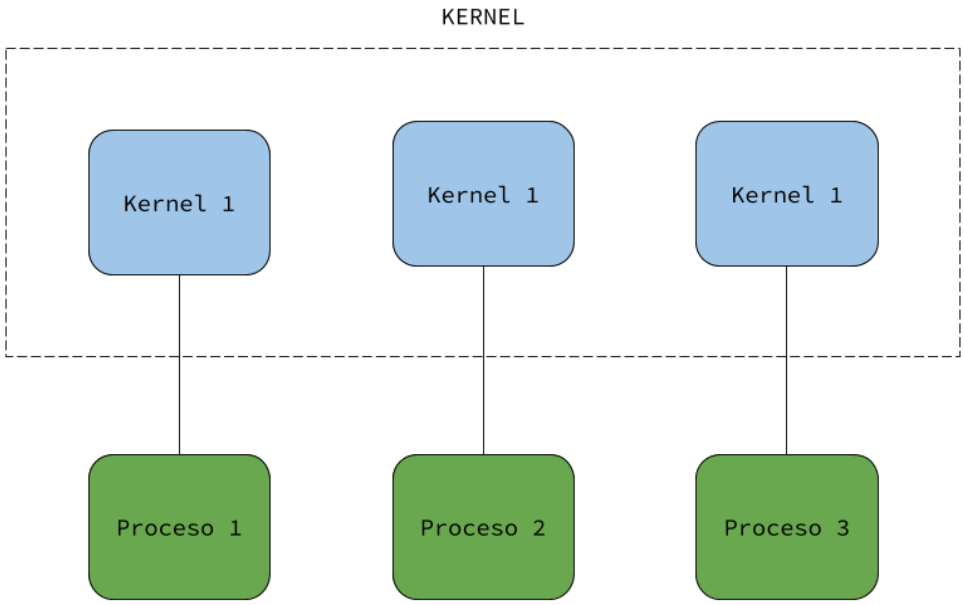
La restricción en los recursos no implica que se le niegue el acceso de dichos recursos a los procesos, mas bien, implica que los procesos no pueden acceder directamente a los recursos. En el argumento anterior se basa el concepto del sistema operativo, el cual se encarga de proveer a los procesos el accesos de los recursos. Es por ello que debe existir una comunicación entre los procesos de usuario y del sistema operativo.

La forma más sencilla de pensar en un sistema operativo es suponer que el Kernel es un único proceso monolítico que administra todos los recursos. La ventaja de la idea anterior es que solamente tenemos que administrar un proceso de Kernel, el siguiente diagrama muestra la estructura básica de como sería un sistema con dicha característica:



Podemos observar que al tener un solo proceso como Kernel, los procesos de usuario deben realizar peticiones a dicho proceso si quieren acceder funcionalidades no permitidas en modo usuario. Lo anterior tiene el inconveniente de que si un proceso de usuario realiza una petición que tardará mucho en ser atendida, entonces los demás procesos que quieran realizar peticiones al Kernel deben esperar. Si las peticiones se hacen sobre el mismo recurso no hay mas opción que esperar, pero cuando se hacen peticiones a diferentes recursos, esos procesos no tendrían la necesidad de esperar.

Una solución que emplean casi todos los sistemas operativos modernos, incluido Pintos, es dividir al Kernel en diferentes procesos. En el caso de Pintos el Kernel consta de hilos en lugar de procesos (veremos la diferencia entre ambos mas tarde). La idea general es tener un hilo principal de Kernel, cuando se necesita un nuevo proceso de usuario, se crea un hilo de Kernel que se asocia al proceso de usuario. Dicho hilo de Kernel atenderá solamente las peticiones que su proceso de usuario genere. El siguiente diagrama muestra la idea general:



Hasta el momento hemos utilizado los conceptos proceso e hilo de forma casi indistinta, como si fueran sinónimos. Sin embargo existe una gran diferencia entre ambos conceptos. Existen múltiples definiciones de hilos y procesos, probablemente cada una de ellas sea correcta, pues cada concepto depende mucho de la implementación del sistema operativo. Esencialmente la diferencia es que los hilos comparten entre sí más recursos que los procesos. En el caso particular de Pintos, los hilos comparten entre ellos el mismo espacio virtual de memoria; por el contrario los procesos tienen un espacio de memoria independiente entre ellos.

2.2. Procesos de usuario en Pintos

En Pintos los procesos de usuario no tienen una estructura de alto nivel asociada, a diferencia de los hilos (*struct thread*). Lo anterior ocurre por el esquema que se utiliza de asociar un hilo de Kernel a un proceso de usuario. Sin embargo si existen funcionalidades dentro del Kernel que están definidas únicamente para los procesos de usuario.

Las funcionalidades asociadas a los procesos de usuario están implementadas en el directorio *src/userprog*. La primer funcionalidad por atender es crear un proceso de usuario, dicha funcionalidad se implementa en el archivo *src/userprog/process.c* por medio de la función *process_execute*:

```
1 tid_t
2 process_execute (const char *file_name)
3 {
4   char *fn_copy;
```

```
5   tid_t tid;
6
7   fn_copy = palloc_get_page (0);
8   if (fn_copy == NULL)
9     return TID_ERROR;
10  strlcpy (fn_copy, file_name, PGSIZE);
11
12  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
13  if (tid == TID_ERROR)
14    palloc_free_page (fn_copy);
15  return tid;
16 }
```

La parte importante de la función ocurre en la línea 12, en donde se invoca a la función *thread_create*. Dicha función se encarga de crear un nuevo hilo de Kernel (para mas detalles ver el sección 1.2), después, el hilo creado ejecuta la función *start_process*. La función *start_process* es la encargada de crear el nuevo proceso de usuario:

```
1  static void
2  start_process (void *file_name_)
3  {
4    char *file_name = file_name_;
5    struct intr_frame if_;
6    bool success;
7
8    /* Initialize interrupt frame and load executable. */
9    memset (&if_, 0, sizeof if_);
10   if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
11   if_.cs = SEL_UCSEG;
12   if_.eflags = FLAG_IF | FLAG_MBS;
13   success = load (file_name, &if_.eip, &if_.esp);
14
15   /* If load failed, quit. */
16   palloc_free_page (file_name);
17   if (!success)
18     thread_exit ();
19
20   asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
21   NOT_REACHED ();
22 }
```

Al igual que los hilos, la parte mas importante de un proceso de usuario es su entorno de ejecución. Como se vio anteriormente, el entorno de ejecución de un proceso está denotado por los valores de los registros del procesador. En la línea 5 se define una estructura de tipo *struct intr_frame*, la cual contiene el valor de los registros del procesador del nuevo proceso. En la línea 13 se invoca la función *load*, que se encarga de cargar el programa del almacenamiento secundario y obtener las direcciones donde se encuentran la pila de ejecución (*esp*) y el segmento de código (*eip*). El último paso consiste modificar el *stack pointer* para que apunte al *stack frame*, y después invocar la función *intr_exit* (línea 20). La función *intr_exit* está programada en ensamblador en el archivo *src/threads/intr-stubs.S* y se muestra a continuación:

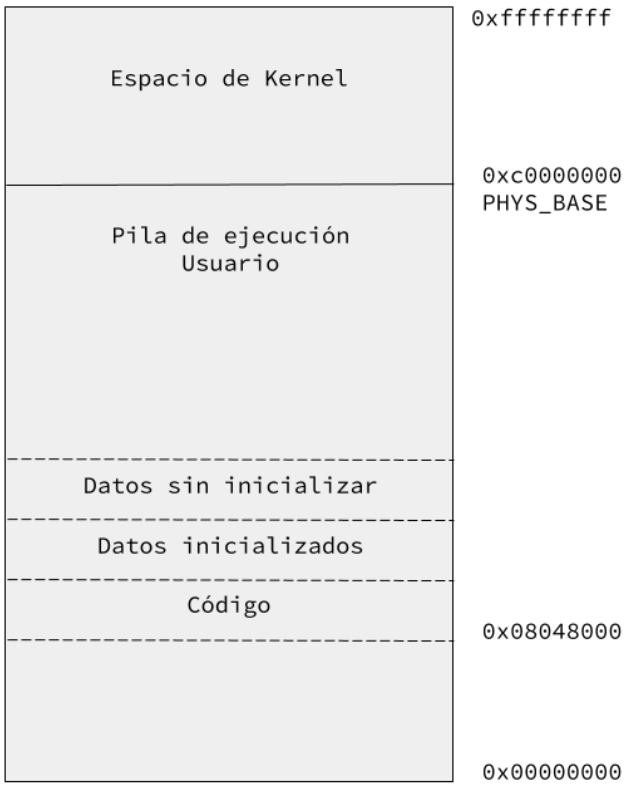
```
1  intr_exit:
2
3    popal
4    popl %gs
5    popl %fs
6    popl %es
7    popl %ds
8
9    addl $12, %esp
10
11     /* Return to caller. */
12   iret
13 .endfunc
```

La idea es simular el regreso de una interrupción. El primer paso es recuperar los registros del procesador de la pila (de la línea 3 a la línea 7), esto es posible porque en la función *start_process* se modificó el valor del *stack pointer* para que apuntara el nuevo *intr_frame* (línea 20 de *start_process*). En la línea 9 se descartan los valores extras de la estructura y después se ejecuta la instruccción *IRET* (línea 12). Según Intel[1], la instrucción *IRET* tiene el siguiente comportamiento:

> Regresa el control desde una excepción o manejador de interrupción a un programa o procedimiento que fue interrumpido por una excepción, interrupción externa o una interrupción generada por software. Si la bandera NT (del registro EFLAGS) no está colocada, la instrucción ejecuta un retorno
>
> lejano de un procedimiento de interrupción. La instrucción IRET retira de la pila de ejecución los valores EIP, CS y EFLAGS y los coloca en los respectivos registros.

2.3. Espacio de usuario

Como se mencionó anteriormente, la arquitectura x86 no cuenta con un mecanismo de acceso directo a la memoria física cuando el procesador se encuentra en modo Virtual. Lo anterior aplica tanto para procesos de usuario como para hilos de Kernel, por lo cual, es necesario definir dentro del espacio virtual una serie de regiones y límites. En el caso de Pintos el espacio virtual se segmenta de la siguiente forma:



El espacio se divide en dos segmentos principales: espacio de usuario y espacio de Kernel, esta dos regiones se dividen por la dirección PHYS_BASE (0xc0000000 o 3GB). La parte baja corresponde al espacio virtual del proceso de usuario y la parte alta corresponde al espacio del Kernel. Es necesario señalar que el espacio de Kernel es general para todos los hilos del Kernel, mientras que el espacio de usuario es independiente para cada proceso de usuario. Por lo tanto, es necesario puntualizar que el espacio del usuario cambia cuando un proceso diferente se ejecuta.

2.4.

sectionSystem Calls

Como se mencionó anteriormente, un sistema operativo debe proporcionar un mecanismo que permita a los procesos de usuario solicitar operaciones que solamente están permitidas para el kernel. El objetivo es que los procesos de usuario no tengan acceso directo a operaciones que pueden poner en riesgo la integridad del sistema operativo. Dicho mecanismo se conoce como llamadas al sistema y solamente se pueden implementar cuando tenemos soporte de hardware. En resumen se necesita un mecanismo para que un proceso de usuario ceda el control del procesador al sistema operativo, lo anterior implica que exista un cambio de privilegios de ejecución.

En el caso de la arquitectura Intel x86, contamos con las interrupciones generadas por software. Una interrupción por software es similar a una interrupción por hardware con la diferencia que la interrupción por software no la genera un dispositivo sino más bien se genera por una instrucción del procesador. En el caso de la Intel x86[1] la instrucción *INT* nos permite realizar los siguiente:.

> La instrucción *INT N* genera una llamada al manejador de interrupciones con el operando de destino (*N*). El operando de destino especifica un número de vector de interrupción de 0 a 255. Cuando el procesador se encuentra en modo virtual, el registro IOPL determina la acción de la instrucción *INT*. Si el registro IOPL es menor a 3, el procesador genera una excepción de protección general (GP). Si el registro IOPL es 3, el procesador ejecuta una interrupción de modo protegido hacia modo privilegiado de nivel 0 (modo kernel).

Como se menciona en el manual de Intel, el parámetro de la instrucción INT corresponde al vector de interrupción, el cual es un número que identifica la interrupción. Para que el sistema operativo pueda ejecutar el manejador correspondiente a cada interrupción, es necesario asociar el número de interrupción con el manejador. En el caso de las llamadas a sistema en Pintos, la asociación se hace en la función *syscall_init* en el archivo *src/userprog/syscall.c*:

```
1 void
2 syscall_init (void)
3 {
4   intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
5 }
```

El siguiente paso a resolver es como pasar los argumentos de la llamada a sistema al sistema operativo. La forma más sencilla de hacerlo es colocar los argumentos en una región de la memoria y que el sistema operativo los obtenga de dicha región. La mayoría de los sistemas operativos utiliza la pila de ejecución del proceso de usuario para almacenar información que después el sistema operativo utilizará. Utilizando la estrategia anterior, el proceso de usuario debe colocar los argumentos en la pila de ejecución antes de generar la interrupción, además, los argumentos deben colocarse en un orden específico.

Usualmente los sistemas operativos proporcionan bibliotecas que colocan los argumentos de las llamadas al sistema y generan la interrupción. Lo anterior se hace con el propósito de que el proceso de usuario no tenga que conocer de antemano el formato con el que se colocan los argumentos. En el caso de Pintos la biblioteca para invocar una llamada a sistema se encuentra en el archivo *src/lib/user/syscall.c*. Lo que se hace es definir una función por cada llamada a sistema, a continuación se muestra el código para la llamada *exit*:

```
1
2 void
3 exit (int status)
4 {
5   syscall1 (SYS_EXIT, status);
6   NOT_REACHED ();
7 }
8
9 #define syscall1(NUMBER, ARG0)                                          \
10     ({                                                                  \
11       int retval;                                                       \
12       asm volatile                                                      \
13         ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
14            : "=a" (retval)                                            \
```

```
15              : [number] "i" (NUMBER),                                    \
16                [arg0] "r" (ARG0)                                         \
17              : "memory");                                                \
18        retval;                                                           \
19      })
```

Podemos observar que la función *exit* solo se define para que el proceso de usuario pueda invocar de forma natural la función. Lo que en realidad sucede es que la función *exit* utiliza un macro que contiene instrucciones de ensamblador que se encargan de colocar los argumentos en la pila de ejecución. Es necesario notar que antes de colocar los argumentos se coloca el identificador de la llamada a sistema, con la finalidad de que el sistema operativo sea capaz de identificar de qué llamada se trata.

Una vez que se genera la interrupción, se invoca el manejador correspondiente. En el caso de Pintos el manejador de llamadas a sistema se encuentra en el archivo *src/userprog/syscall.c*, en la función *syscall_handler*. La función *syscall_handler* recibe como parámetro una estructura de tipo *intr_frame*, definida en el archivo *src/threads/interrupt.h*:

```
1  struct intr_frame
2    {
3      uint32_t edi;               /* Saved EDI. */
4      uint32_t esi;               /* Saved ESI. */
5      uint32_t ebp;               /* Saved EBP. */
6      uint32_t esp_dummy;         /* Not used. */
7      uint32_t ebx;               /* Saved EBX. */
8      uint32_t edx;               /* Saved EDX. */
9      uint32_t ecx;               /* Saved ECX. */
10     uint32_t eax;               /* Saved EAX. */
11     uint16_t gs, :16;           /* Saved GS segment register. */
12     uint16_t fs, :16;           /* Saved FS segment register. */
13     uint16_t es, :16;           /* Saved ES segment register. */
14     uint16_t ds, :16;           /* Saved DS segment register. */
15
16     uint32_t vec_no;            /* Interrupt vector number. */
17
18     uint32_t error_code;        /* Error code. */
19
20     void *frame_pointer;        /* Saved EBP (frame pointer). */
21
22     void (*eip) (void);         /* Next instruction to execute. */
23     uint16_t cs, :16;           /* Code segment for eip. */
24     uint32_t eflags;            /* Saved CPU flags. */
25     void *esp;                  /* Saved stack pointer. */
26     uint16_t ss, :16;           /* Data segment for esp. */
27   };
```

Como podemos observar, la estructura contiene el valor de los registros del proceso de usuario al momento de generarse la interrupción. A partir de ésta estructura se puede obtener el valor de *stack pointer* y por consiguiente se puede acceder a los argumentos colocados en la pila de ejecución del proceso de usuario, así como el identificador de la llamada a sistema.

Es necesario señalar que con los valores de la estructura se reconstruye el entorno de ejecución del proceso de usuario, una vez que la interrupción terminar. Por ello, se debe tener especial cuidado al modificar el valor de alguno de los campos de la estructura. Sin embargo, en algunos casos es necesario modificar los valores de la estructura, por ejemplo, el valor de retorno de cualquier llamada, se debe almacenar en el registro EAX.

3. Forma de Trabajo

Pasos a seguir para trabajar en esta práctica:

1. Tienes que aceptar la asignación Pintos Codebase User Programs en Github Classroom.
2. A partir de la rama `main` crea una nueva rama llamada `proyecto02/process-management-syscalls` y posiciónate en ella. Es importante que la rama base de este trabajo no contenga código de la solución del proyecto 01 que está compuesto por las tres prácticas anteriores.
3. En el repositorio reemplaza el archivo `.github/workflows/run-tests.yml` con el que entregamos junto con este documento.
4. Haz commit de estos cambios y sube (*push*) la rama al repositorio.
5. Crea un *Pull Request* para la rama `proyecto02/process-management-syscalls` hacia la rama `main`. Este *Pull Request* debe de contener un archivo en la sección de *Files changed*, que es exactamente el que introdujiste en los pasos anteriores.
6. En la sección de *Actions* solamente vas a pasar las pruebas de *Argument Passing*, pero tienes que implementar las llamadas a sistema *exit*, *exec* y *wait* para pasar el resto de las pruebas.
7. Una nota importante es que para pasar las pruebas vamos a utilizar la máquina virtual `qemu`, pues `bochs` puede tener algunos problemas.

En el archivo `readme.md` del repositorio se incluyen instrucciones de cómo compilar el código y de cómo ejecutar algunas pruebas. Para ejecutar todas las pruebas de esta práctica, puedes acceder al contenedor, posicionarte en el directorio `src/userprog` y ejecutar la instrucción
`bash execute-tests-processes-syscalls`.

4. Actividades

Implementar las llamadas a sistema *exec*, *wait* y *exit*. El comportamiento de cada una de las llamadas se muestra a continuación:

int exec(const char* cmd)

Ejecuta el programa cuyo nombre se encuentra contenido en la cadena CMD , la cual también contiene los argumentos separados por espacios. La llamada debe regresar el identificador de proceso, en caso de que el programa se haya cargado correctamente, en caso contrario regresa -1. El proceso que ejecuta la llamada no debe continuar hasta que conoce el estado de carga del proceso hijo, por lo cual, es necesario sincronizar ambos procesos.

int wait(int pid)

El proceso que invoca la llamada se bloquea hasta que el proceso identificado con el identificado de proceso PID termina su ejecución. El proceso PID debe ser un proceso creado con la llamada *exec* por el proceso que invoca *wait*. El valor de retorno de *wait* debe ser el valor con el que el proceso PID terminó.

Si el proceso PID terminó su ejecución antes de la llamada *wait*, el proceso padre no se bloquea pero debe recibir el valor de salida. La llamada *wait* solo tiene efecto en hijos directos y si se invoca mas de una vez sobre el mismo PID las llamadas subsecuentes no deben bloquear al proceso y deben regresar -1.

El hilo principal de Pintos ejecuta la función *process_wait* (archivo process.c) para esperar al proceso que ejecuta las pruebas, por el momento dicha función solamente duerme al proceso por 200 ticks mediante la función *timer_sleep*. Es recomendable implementar la llamada *wait* en la función *process_wait* y después invocarla en *syscall_handler*.

void exit(int status)

Termina el proceso actual de usuario, regresando su *status* al hilo de Kernel que tiene asociado. En el caso de que el proceso padre haya ejecutado *wait* sobre el proceso, el valor *status* se le regresa. Antes de terminar, el proceso debe imprimir un mensaje con el siguiente formato:

```
1 printf ("%s: exit(%d)\n", name, status);
```

4.1. Implementación (5 ptos)

- exec-once
- exec-arg
- exec-missing
- exec-multiple
- exit
- wait-simple
- wait-twice
- wait-bad-pid

4.2. Pruebas Extra (3 pts)

- wait-killed
- exec-bound
- exec-bound-2
- exec-bound-3
- exec-bad-ptr

Para poder resolver estas pruebas será necesario implementar el acceso seguro a memoria desde una llamada a sistema. Para ejecutar las pruebas referentes a los puntos extra es necesario descomentar la línea 4 del archivo `src/userprog/execute-tests-processes-syscalls`.

4.3. Documento de Diseño (5 pts)

Contestar todas las secciones del documento de diseño `designdocs/userprog/designdoc-processes-syscalls.md`.

5. Consejos

Para la llamada `exec` es necesario invocar la función `process_execute` que se encarga de cargar el proceso. El único detalle es que no hay una sincronización para que el proceso padre pueda determinar si el proceso hijo se cargó correctamente, esto es requerido en la prueba `exec_missing`. Se recomienda utilizar un semáforo inicializado en cero, el proceso padre debe hacer `sema_down` justo después de invocar `thread_create`, el proceso hijo debe notificar si se cargó correctamente al padre y hacer `sema_up`, esto debe ser justo después de invocar la función `load` en la función `start_process`.

Cada proceso debe llevar registro de los procesos que crea mediante la *system call* `exec`. Para ello se recomienda utilizar una lista donde los elementos de la misma deben ser estructuras que contengan al menos una forma de acceder a la estructura `struct thread` del proceso hijo. También se recomienda que la estructura contenga un campo para almacenar el valor de salida del hijo. Dichas estructuras deben ser inicializadas utilizando `malloc` pues de esta forma se evitan problemas cuando algún proceso hijo termina antes de que el proceso padre trate de hacer `wait` sobre él.

Para la llamada `wait` se recomienda utilizar el mismo esquema del semáforo que se utiliza para esperar la carga del proceso hijo, con la diferencia de que se debe hacer `sema_up` en el momento en el que el proceso hijo invoca la llamada `exit`.

6. Entregables

En Google Classroom agrega el enlace del `Pull Request` que contiene tu solución y en donde se deben de observar los resultados de Github Actions al correr las pruebas. También marca la tarea como entregada en Google Classroom.