Práctica 5: Paso de argumentos

David Flores Peñaloza (dflorespenaloza@gmail.com)
José Nieves Morán (nieves@ciencias.unam.mx)
Angel Renato Zamudio Malagón (renatiux@gmail.com)

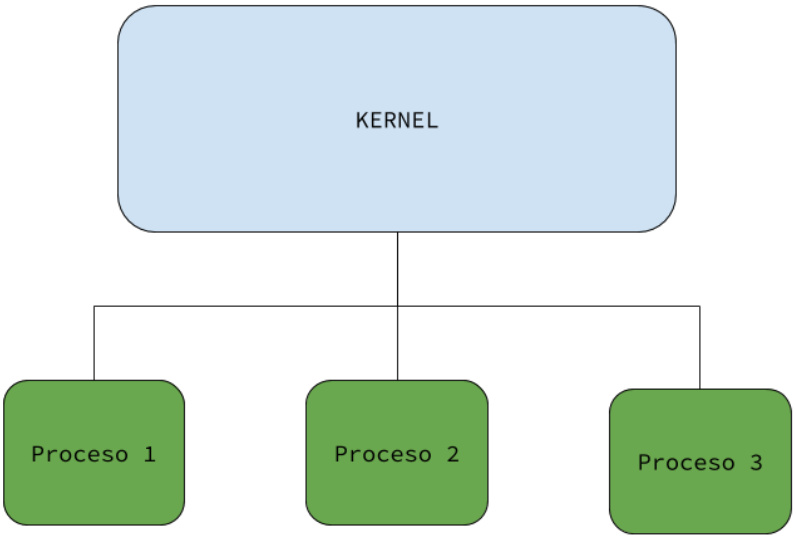
Fecha de entrega: Lunes 30 de marzo de 2020

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.

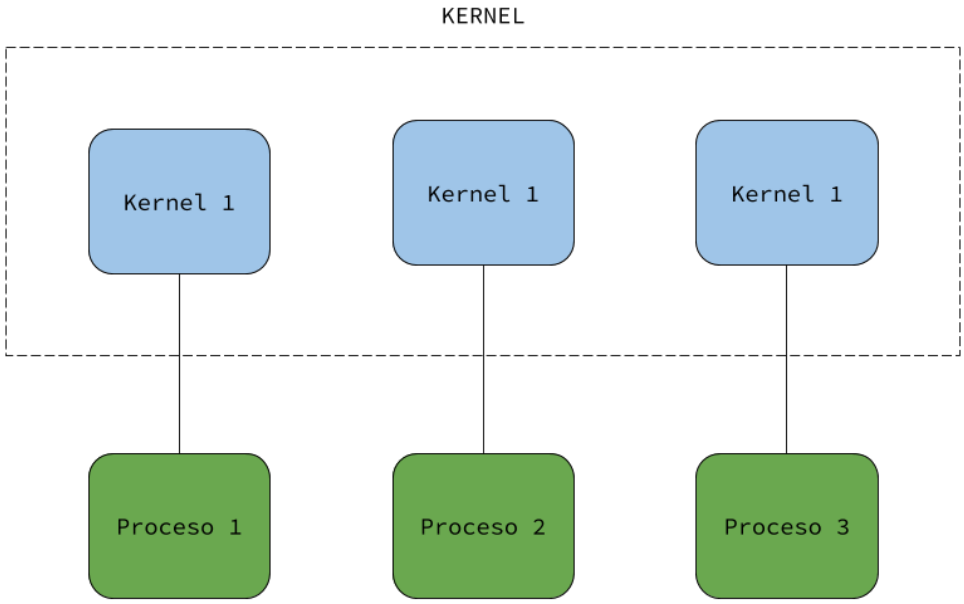
La forma más sencilla de pensar en un sistema operativo es suponer que el Kernel es un único proceso monolítico que administra todos los recursos. La ventaja de la idea anterior es que solamente tenemos que administrar un proceso de Kernel, el siguiente diagrama muestra la estructura básica de como sería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

1.1. 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
```

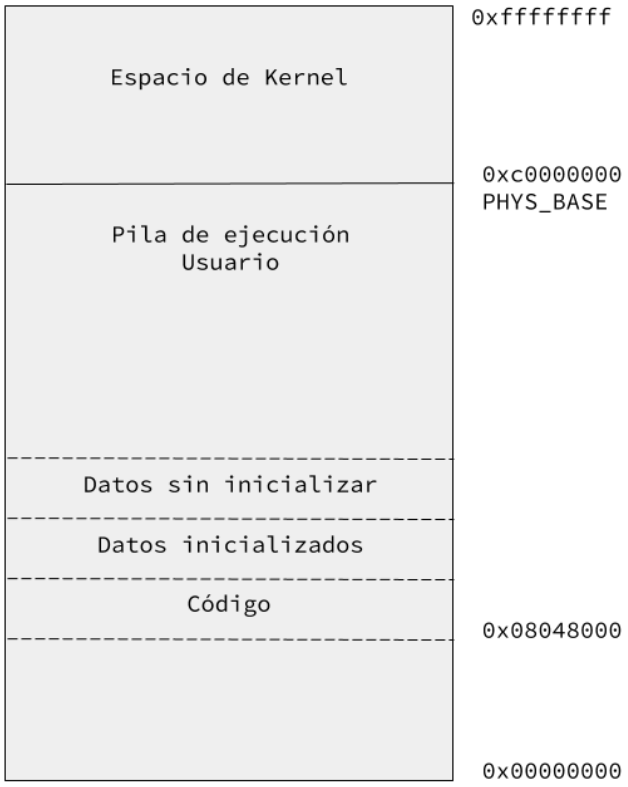
La idea es simular el regreso de una interrupción. El primer paso es recuperar los registros del procesador de la pila (de la línea 3 a la línea 7), esto es posible porque en la función *start_process* se modificó el valor del *stack pointer* para que apuntara el nuevo *intr_frame* (línea 20 de *start_process*). 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.

1.2. 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. Requerimientos

Implementar el paso inicial de argumentos de la línea de comandos al *stack* del proceso de usuario. En la función *process_execute* el parámetro *file_name* no solo contiene el nombre del archivo, también contiene los argumentos que se pasan por la línea de comandos. Por ejemplo si ejecutamos **pintos run 'echo x y'** el parámetro *file_name* contendrá la cadena 'echo x yz'. El formato que deben tener los argumentos en el *stack* se muestra a continuación suponiendo que la línea de comandos es 'echo x yz'.

DIRECCIÓN	ELEMENTO	VALOR	TIPO
0xbffffffd	arg[2][...]	″yz\0″	char[3]
0xbffffffb	arg[1][...]	″x\0″	char[2]
0xbffffff6	arg[0][...]	″echo\0″	char[5]
0xbffffff5	word-align	0	uint8_t
0xbffffff1	argv[3]	0	char*
0xbffffed	argv[2]	0xbffffffd	char*
0xbffffe9	argv[1]	0xbffffffb	char*
0xbffffe5	argv[0]	0xbffffff6	char*
0xbffffe1	argv	0xbffffe5	char**
0xbffffed	argc	3	int
0xbffffe9	return address	0	void(*)()

El contenido de los argumentos se tienen que colocar en formato de arreglo de cadenas, en la direcciones más altas del *stack* se debe colocar el contenido del arreglo, es decir, las cadenas. Es necesario coloar el caracter de fin de cadena *\0* y empezar de abajo (direcciones menores) hacia arriba (direcciones mayores) de tal forma que el caracter de fin de cadena quede en la dirección más alta de toda la cadena.

El valor del *stack pointer* es muy importante pues por medio de este valor el proceso de usuario accede a los argumentos. En el ejemplo anterior el *stack pointer* debe ser inicializado con el valor 0xbffffe9. Para acceder al *stack pointer* del proceso de usuario se utiliza una estructura de tipo *struct intr_frame*, la cual se obtiene en la función *start_process* y es posible acceder al *stack pointer* a través del campo *esp* de dicha estructura.

3. Pruebas

- args-none
- args-single
- args-multiple
- args-many
- args-dbl-space

Si únicamente realizamos el paso de argumentos de la línea de comandos al *stack*, la pruebas no pasarán pues todas ocupan dos llamadas a sistema que no se encuentran implementadas. En pintos el manejador de llamadas a sistema se encuentra en el archivo *src/userprog/syscall.c*, el cual se encuentra vacío. Para que las pruebas pasen en necesario implementar de forma temporal las llamadas a sistema *write*, *exit* y *wait*, para ello la función *syscall_handler* se debe implementar de la siguiente forma:

```
1  static void
2  syscall_handler (struct intr_frame *f UNUSED)
3  {
4    uint32_t* esp = f->esp;
5    uint32_t syscall = *esp;
6    esp++;
7
8    switch(syscall) {
9      case SYS_WRITE: {
10       int fd = *esp;
11       esp++;
12       void* buffer = (void*)*esp;
13       esp++;
14       unsigned int size = *esp;
15
16       putbuf(buffer, size);
17
18       break;
19     }
```

```
    case SYS_EXIT: {
      printf("%s: exit(0)\n", thread_current()->name);
      thread_exit ();
      break;
    }
  }
}
```

También es necesario reimplementar temporalmente la función *process_wait*, pues al ejecutar una prueba de usuario en Pintos, el thread principal ocupa la función *process_wait* para poder apagar el sistema solamente hasta que la prueba ha finalizado. La solución temporal es implementar la función *process_wait* como se muestra a continuación.

```
int
process_wait (tid_t child_tid UNUSED)
{
  timer_sleep(200);

  return -1;
}
```

De momento no se explicarán los códigos anteriores pues son soluciones temporales para que las pruebas que verifican el paso de argumentos funcionen correctamente. Más adelante se explicará a detalle el mecanismo de las llamadas a sistema.

4. Consejos

El momento de realizar el paso de argumentos en el ciclo de creación del proceso de usuario es muy importante, se recomienda hacerlo en la función *start_process* justo después de que se invoca a la función *load* y que el valor devuelto por dicha función sea *true*. También es recomendable definir una función que se encargue únicamente de colocar los argumentos e invocarla desde *start_process*.

Para colocar el contenido de los argumentos se recomienda usar un apuntador de tipo *uint8_t* inicializado en la dirección 0xbfffffff y colocar caracter por caracter todos los arguméntos, cada que se coloca un nuevo caracter el apuntador se decrementa en 1 y se coloca el valor usando el operador de desreferenciación (*), al final de cada cadena se debe colocar el catacter de fin de cadena (\0). Para colocar los apuntadores se recomienda recorrer el *stack* partiendo de la última dirección donde se colocó un caracter hasta la dirección 0xbfffffff e identificar el inicio de cada uno de los argumentos, para colocar la dirección de cada argumento se recomienda utilizar un apuntador de tipo *uint32_t*.

Es necesario considerar el caso en el que la línea de comando contiene varios espacios entre un argumento y otro, los espacios extras deben ser omitidos y considerarse como si solamente hubiera un espacio.

Para poder saber si estamos colocando de forma adecuada los argumentos se puede utilizar la siguiente función para imprimir el contenido de una región específica de la memoria. Si se quieren imprimir la primeras 10 direcciones a partir de PHYS_BASE se tiene que invocar así: *print(PHYS_BASE - 10, PHYS_BASE)*

```
void
print(void* begin, void* end) {
  void* current = end;

  printf("address \t integer \t char\n");
  while(current >= begin) {
    printf("%p \t %d \t %c\n", current, *(char*)current, *(char*)current);
    current--;
  }
}
```

5. Preguntas

1. ¿Por qué el espacio virtual de memoria del Kernel esta junto al espacio virtual del proceso de usuario?

2. ¿De que forma se garantiza que un proceso de usuario solamente pueda acceder a su propia memoria?

3. ¿Por qué es el sistema operativo el encargado de colocar los parámetros iniciales de un proceso?

6. Entregables

La práctica se debe enviar al correo **renatiux@gmail.com** con el asunto **PRÁCTICA 5**, se debe de enviar un archivo comprimido tar.gz el cual debe contener los archivos que se modificaron y un archivo README.txt. Por ejemplo si modificaron los archivos *timer.c* y *thread.h* la estructura del archivo debe ser:

```
|— README.txt
|— src
  |— devices
     |— timer.c
  |— threads
     |— thread.h
```

El archivo README.txt debe contener el nombre completo de los integrantes del equipo y las respuestas de las preguntas.