Trabajo Práctico 2

Instituto Tecnológico de Buenos Aires - Sistemas Operativos (72.11)

Grupo 19

Ignacio Searles	**Augusto Barthelemy Solá**	**Santiago Bassi**
isearles@itba.edu.ar	abarthelemysola@itba.edu.ar	sabassi@itba.edu.ar
64.536	64.502	64.643

11 de noviembre de 2024

Resumen

El presente informe trata sobre el desarrollo de un kernel que administra los recursos de hardware de una computadora y que tiene una API para interactuar con el espacio de usuario. En el espacio de usuario se desarrolló un shell que permite ejecutar diferentes módulos cuyo objetivo es mostrar el funcionamiento del sistema.

1 Memory Manager

Ambos memory managers, al momento de su inicialización, reciben la cantidad de memoria que van a administrar. De esa memoria, consumen una parte para almacenar los datos necesarios para su funcionamiento.

Para los tests del memory manager, se decidió que cuando se compila fuera del kernel, este corra indefinidamente, ya que se puede terminar la ejecución matando el proceso desde la shell. En cambio, cuando se compilan y ejecutan los tests dentro del kernel, se decidió que corran una cantidad determinada de veces, aún así se puede detener la ejecución del mismo mediante un kill del proceso de testeo.

1.1 Bitmap Memory Manager

Para el bitmap, se decidió usar dos bits para representar cada bloque de memoria, lo que permite utilizar tres estados (FREE, USED, START). A cada bloque se le asignó un tamaño definido por la macro BLOCK_SIZE.

A continuación, se detallan las instrucciones para compilar y ejecutar los tests del bitmap memory manager:

1. Compilación y ejecución de los tests dentro del kernel:
   (a) Tener compilado el Toolchain. Dentro de la carpeta Toolchain, ejecutar **make**.
   (b) En la raíz del proyecto, ejecutar **make**.
   (c) Luego, ejecutar **./run.sh**.
   (d) En la shell que se abre, ejecutar **test_mm**.
2. Compilación y ejecución de los tests fuera del kernel:

(a) En la raíz del proyecto, ejecutar **make bitmaptest**.

(b) Luego, ejecutar **cd Testing**.

(c) Por último, ejecutar **./bitmapTest <memoryAmount>**.
Siendo memoryAmount la cantidad de memoria que se desea asignar.

1.2 Buddy Memory Manager

Para el buddy, se decidió almacenar la información en un árbol, donde cada nodo almacena uno de los siguientes estados: FREE, SPLIT o USED. Al no tener memoria dinámica para generar el árbol, se decidió almacenarlo en un bloque de memoria contigua, disponiendo sus elementos en orden preorder.

A continuación, se detallan las instrucciones para compilar y ejecutar los tests del buddy memory manager:

1. Compilación y ejecución de los tests dentro del kernel:

   (a) Tener compilado el Toolchain. Dentro de la carpeta Toolchain, ejecutar **make**.

   (b) En la raíz del proyecto, ejecutar **make buddy**.

   (c) Luego, ejecutar **./run.sh**.

   (d) En la shell que se abre, ejecutar **test_mm**.

2. Compilación y ejecución de los tests fuera del kernel:

   (a) En la raíz del proyecto, ejecutar **make buddytest**.

   (b) Luego, ejecutar **cd Testing**.

   (c) Por último, ejecutar **./buddytest <memoryAmount>**.
   Siendo memoryAmount la cantidad de memoria que se desea asignar.

2 Process Manager

El process manager proporciona varias funciones importantes para la gestión de procesos.
`init_process_manager` inicializa el gestor de procesos. `create_process` permite crear un proceso. `exit_process` finaliza un proceso con un código de estado, pero no lo borra de la tabla de procesos, mientras que `kill_process` hace algo similar, pero lo borra de la tabla de procesos mediante `remove_process`. `block_process` y `unblock_process` controlan su estado de ejecución. Para obtener información sobre los procesos en ejecución, se pueden utilizar funciones como `get_processes`, `get_num_processes`, `get_max_pid` y `get_ps_data`. Además, `wait_process` permite que un proceso espere a la finalización de un proceso hijo, y `nicent` ajusta la prioridad de un proceso.

En la creación de procesos armamos el stack de tal manera que al ejecutarse el próximo context switch sea análogo a un proceso que ya estaba ejecutando. Durante la creación del proceso seteamos el registro `rip` para saltar a una función wrapper, la misma se encarga de llamar a la función dada con `argv` y `argc` (tanto el puntero a la función, pid y argv usados por el wrapper se cargan en el stack). Además, la función wrapper hace un exit del proceso al terminar la ejecución del mismo.

En caso que algún proceso provoque una excepción, se maneja la excepción y se realiza un `kill_process` del proceso.

3 Scheduler

El scheduler es el encargado de gestionar todo lo relacionado con los cambios de contexto. `init_scheduler` inicializa el scheduler. Para gestionar los procesos, se utiliza `schedule_process` para agregar un proceso al scheduler, y `deschedule_process` para removerlo. `get_current_process` devuelve el proceso actualmente en ejecución. La función `change_process_priority` permite cambiar al proceso de lista. Finalmente, `context_switch` realiza el cambio de contexto entre procesos, permitiendo la alternancia en la ejecución de distintos procesos.

Para el scheduler, se decidió usar un vector con listas; cada posición del vector representa una prioridad distinta, y en cada lista se guardan los procesos con esa prioridad. Las prioridades son `HIGH, MEDIUM, LOW`. Al momento de hacer un `context_switch`, se elige el siguiente proceso a ejecutar de manera pseudoaleatoria. Se decidió que un proceso con prioridad `HIGH` tiene el doble de probabilidades de ser elegido que un proceso con prioridad `MEDIUM`, y uno con prioridad `MEDIUM` tiene el doble de probabilidades de ser elegido que uno con prioridad `LOW`. Para garantizar que se cumpla esta distribución de probabilidad (teniendo en cuenta la cantidad de procesos de cada prioridad), se planteo el siguiente sistema de ecuaciones.

Sean $h, m, l \in [0; 1]$ las probabilidades de ejecutar un proceso de dicha prioridad.

$$h = 2m$$

$$m = 2l$$

Sean $h\_len, m\_len, l\_len \in \mathbb{N}$ las longitudes de las listas de prioridad.

$$1 = h * h\_len + m * m\_len + l * l\_len$$

Resolviendo el sistema se obtiene,

$$h = \frac{4}{4h\_len + 2m\_len + l\_len}$$

$$m = \frac{2}{4h\_len + 2m\_len + l\_len}$$

$$l = \frac{1}{4h\_len + 2m\_len + l\_len}$$

Si h_len, m_len, l_len son 0, las mismas formulas todavía valen. El caso donde todos son 0 no lo consideramos, pues siempre existe el proceso idle.

Luego las probabilidades de correr un proceso de una lista de prioridades particular es,

$$h\_list = h * h\_len = h\_len * \frac{4}{4h\_len + 2m\_len + l\_len}$$

$$m\_list = m * m\_len = m\_len* = \frac{2}{4h\_len + 2m\_len + l\_len}$$

$$l\_list = l * l\_len = l\_len * \frac{1}{4h\_len + 2m\_len + l\_len}$$

Dichas probabilidades las usamos al momento de hacer el context switch para decidir de que lista de prioridades vamos a correr el siguiente proceso.

4 Semaphore Manager

Este módulo es el encargado de gestionar los semáforos mediante las siguientes funciones: `sem_open`, `sem_close`, `sem_up` y `sem_down`. Al utilizar estas funciones, al momento de hacer un `open` se obtiene un ID del semáforo, y luego se pueden ejecutar el resto de las funciones utilizando ese ID. Además, se implementaron semáforos con nombres, permitiendo compartirlos entre procesos. Para ello, se utilizan las funciones `sem_open_named`, `sem_close_named`.

Dado que no tenemos la posibilidad de mapear los semáforos en la memoria de un proceso, se decidió que, si un proceso realiza un `open` de un semáforo que ya existe, no se realice ninguna acción adicional. Por esta limitación, si un proceso conoce el nombre de un semáforo, puede hacer un `up` o `down` sobre él sin necesidad de haberlo abierto previamente.

Relacionado con esta última limitación, se decidió que si un proceso ejecuta un `sem_close` o `sem_close_named`, el semáforo se elimine del vector de semáforos, de modo que ningún otro proceso pueda acceder al mismo posteriormente, si algún proceso estaba bloqueado en ese semáforo el mismo es liberado.

5 Pipes Manager

Este módulo permite la comunicación entre procesos mediante pipes, implementando las funciones `pipe_open`, `pipe_close`, `pipe_write` y `pipe_read`. Al igual que los semáforos, los pipes pueden crearse con nombres mediante la función `pipe_open_named`, lo que permite que se compartan entre procesos. Los pipes se gestionan mediante un vector de punteros, con un máximo de `MAX_PIPES`, y cada pipe se define como una estructura que incluye un buffer circular de tamaño `BUFFER_SIZE`.

Al igual que en los semáforos, no existe una noción de pipe "abierto" dentro del espacio de memoria de un proceso, por lo que cualquier proceso puede acceder a un pipe conociendo su file descriptor.

Para generalizar el comportamiento de los pipes agregamos dos modos de operación a los mismos: `EOF_CONSUMER` y `NON_EOF_CONSUMER`. En el modo `NON_EOF_CONSUMER` al enviar un EOF a un pipe, este deja de ser escribible, permitiendo solo la lectura de los datos en el buffer previos al EOF. En el modo `EOF_CONSUMER`, al enviar un EOF, el read va a leer hasta el EOF y consumirlo, permitiendo realizar nuevos write al pipe.

Decidimos que cuando un proceso esta bloqueado en un read o un write, si el pipe se cierra, el proceso se desbloquea y se le devuelve EOF.

6 Entrada y salida

Para generalizar el comportamiento de la entrada y salida de los procesos, al momento de crear un proceso, se le debe especificar un pipe de entrada y un pipe de salida. A nivel userland, funciones como `puts`, `printf`, `scanf`, y el resto de funciones I/O, leen y escriben a los pipes de entrada y salida del proceso.

Existen dos pipes standard creados al momento de inicializar el Kernel, un pipe de entrada denominado `stdin` y otro de salida denominado `stdout`. Estos pipes presentan comportamiento especial:

- `stdin`: el driver de teclado escribe a este pipe. Es un pipe de tipo `NON_EOF_CONSUMER` por lo que los EOF pueden ser consumidos y el pipe reutilizado.
- `stdout`: existe un proceso creado por el Kernel denominado `screen_service` que imprime el contenido de este pipe a la terminal.

7 Idle

El idle es un proceso que se crea al inicializar el process manager, el mismo no se puede terminar, bloquear, ni cambiar su prioridad. Su función es, si los tuviera, liberar a sus hijo que estén en estado `EXITED` y luego ejecutar yield para ceder la cpu a otro proceso. El mismo al comenzar es el encargado de crear la `shell` y el proceso `screen_service`.

8 Shell

La shell posee comandos que corren como procesos y otros que son comando built-in. Ejecutando el comando help se puede ver una lista con todos los comandos disponibles y si son procesos o built-in. Para todos los módulos que corren como procesos, se les puede poner como último argumento `&` para que corran en background.

La diferencia entre correr un proceso en background o foreground es que si el proceso está en foreground la shell hace un `waitpid` que espera a que el proceso termine y no si está en background. Los procesos pueden elegir si imprimir o no al buffer de texto si están en background.

Agregamos shortcuts para matar procesos corridos por la shell en foreground. `Ctrl + C` mata al proceso actual y `Ctrl + Shift + C` mata al proceso actual y a todos sus descendientes (`superkill`).

Agregamos un shortcut para enviar EOF a `stdin`. `Ctrl + D` envía un EOF al pipe `stdin`.

A continuación se detallan algunos módulos que sirven para testear el funcionamiento del sistema:

Proceso	**Argumento 1**	**Argumento 2**	**Argumento 3**
test_mm	max_iters	max_memory	-
test_processes	max_iters	max_processes	-
test_wait	max_iters	max_processes	-
test_prio	max_processes	-	-
test_prio_dist	max_processes	-	-
test_idle_cleanup	max_processes	-	-
test_sync	max_iters	max_pair_processes	use_syncro
test_pipes	msg_len	max_pair_processes	-
mem	-	-	-
block	pid	-	-
unblock	pid	-	-
kill	pid	-	-
nice	pid	priority	-
loop	waiting_time	-	-
phylo	-	-	-
cat	-	-	-
wc	-	-	-
filter	-	-	-
echo	string	-	-

Tabla 1: Tabla de procesos y sus argumentos

Si bien hay varios test que fueron brindados por la cátedra, decidimos hacer algunos tests propios para probar el funcionamiento del sistema.

- test_priority_dist Este test crea múltiples procesos con diferentes prioridades y cuenta la cantidad de veces que ese proceso es elegido por el scheduler. Luego imprime cuantas veces fueron elegidos los procesos de cada una de las prioridades. Fue creado con el propósito de verificar el correcto funcionamiento del scheduler pseudoaleatorio.
- test_pipes Este test verifica la funcionalidad de los pipes creando pares de procesos escritores y lectores que se comunican a través de un pipe, verificando que la cantidad de bytes escritos sea la misma que la cantidad de bytes leídos.
- test_idle_cleanup Este test crea múltiples procesos y finaliza sin hacer wait a ninguno de ellos, verificando que el proceso idle los limpie correctamente.
- test_wait Este test crea dos grupos de procesos, los mismos tienen un pequeño delay y luego retornan un valor. Finalmente, el test verifica si la suma de los valores de retorno de ambos grupos fue el mismo.

9 Otras decisiones de diseño

- Se decidió crear un archivo `def.h` para definir las constantes y estructuras que se usan tanto del lado del kernel como del userland.
- Se decidió que al momento de crear un proceso el mismo se cree con prioridad `LOW`.
- Se decidió que la syscall `kill` permita especificar si al matar a un proceso se matan también a sus descendientes.
- Se decidió que en la shell, al llamar al comando kill, el mismo reciba el pid del proceso como argumento y de opcionalmente el argumento `kill` para matar a los descendientes del proceso.
- Se decidió que los procesos ejecutados sigan en la lista de PCBs hasta que mueran o terminen y se les haga `wait`. Por dicho motivo la shell tiene un comando `cleanup` para limpiar los procesos `EXITED` a los que no se les hizo `wait`.
- En lugar de tener un único comando en la shell que bloquee o desbloquee procesos dependiendo de su estado, se decidió tener dos comandos distintos `block` y `unblock`. Ambos reciben el pid del proceso a bloquear o desbloquear.
- El driver de teclado tiene dos modos de operación: `CANNONICAL` y `NON_CANONNICAL`. El modo `CANNONICAL` escribe a `stdin` cuando recibe un `newline` e imprime a `stdout` cada vez que se presiona una tecla. El modo `NON_CANNONICAL` no imprime a `stdout` y escribe a `stdin` cada vez que recibe una tecla.

10 Limitaciones

- El sistema tiene un límite de 256 procesos.
- El sistema tiene un límite de 256 semáforos.
- El sistema solo puede manejar tres prioridades.
- El stack de los procesos es de 8KB.
- El sistema cuenta con 4MB de memoria.

- Al correr `Ctrl + Shift + C` o `Ctrl + C`, pueden quedar recursos abiertos, lo cual implica leaks de memoria.

En cuanto a las limitaciones numéricas, el sistema fue diseñado de tal manera que se pueden cambiar fácilmente. En el archivo `def.h` se encuentran las constantes que definen estas limitaciones.

11 Warnings de PVS-Studio

A continuación se detallan los warnings encontrados por PVS-Studio que son considerados falsos positivos:

- Varios Warnings en Bootloader, se decidió no tocar este código.
- Error V576. Este error se encuentra en muchos archivos, el mismo es desestimó ya que indica un error de formato en printf y en scanf, pues PVS-Studio asocia printf y scanf con las funciones de la librería estándar de C.
- Error V776. Este error indica que hay un loop infinito. Pero este error es un falso positivo ya que se utiliza para los tests.
- Error V566. Este error indica que puede haber problemas al utilizar una constante entera como puntero. Esto es un falso positivo ya que en nuestro sistema estamos harcodeando la dirección donde se carga el código de userland a través de esta constante.
- Error V560. Este error nos indica que la condición `if (c != '\n')` siempre es verdadera. Pero analizando el siguiente fragmento de código podemos ver rápidamente que es un falso positivo.

```
while (i > 0) {
    i = 0;
    c = 0;
    while (c != '\n' && sys_read(stdin, &c, 1) != EOF && i < BUF_SIZE - 1)
        buffer[i++] = c;
    buffer[i] = 0;
    sys_write(stdout, buffer, i + 1);
}
return 0;
```