Trabajo Práctico N°2

Construcción del Núcleo de un Sistema Operativo y mecanismos de administración de recursos

v1.0.0

1. Equipo

Nombre	Apellido	Legajo	E-mail
Ignacio Gastón	Frund	61.465	ifrund@itba.edu.ar
Joaquín José	Huerta	64.252	johuerta@itba.edu.ar
Thomas Adrianus	Ruijgt	62.875	truijgt@itba.edu.ar

2. Repositorio

La solución y su documentación se encuentran versionadas en: `TP2-SO-2025Q2`. La branch correspondiente a la entrega final es `main` y el commit es `91b96940c2ee499b48535394701fe46f3b7d94b1`.

3. Decisiones tomadas durante el desarrollo

La mayor parte de estas decisiones se tomaron luego de su consulta correspondiente a la cátedra.

El sistema utiliza hasta 1GB de memoria (indicados en `run.sh`) y los administradores de memoria cuentan con constantes relacionadas a este valor (1019 MB con el administrador simple, 512MB con el administrador buddy para evitar conflictos con los espacios de del BareBones, Kernel y Userland). Esto se decidió porque WSL permite como máximo 1GB por default y el sistema en sí es lo suficientemente simple como para que 1GB sea más que suficiente.

De todas formas, el sistema BareBones utilizado es de 64 bits, por lo que podría utilizarse mucha más memoria si se deseara.

Se decidió que el proceso `idle` cumpla también las funciones de `init` para ahorrar un proceso que parece innecesario. Por lo tanto, `idle` crea la `shell` y también recibe a cualquier proceso huérfano.

El comando `nice` no puede afectar la prioridad de `idle`, ya que se determinó innecesario aumentar la prioridad de un proceso que solo hace yield.

El enunciado pide que el comando `ps` muestre si el proceso está en foreground, pero luego de consultarlo por mail se determinó que no es necesario incluir este dato ya que la implementación de foreground y background se encuentra en Userspace y no en Kernel, donde se encuentra `ps`.

En la implementación de `kill` se decidió que es posible matar procesos bloqueados, y como un proceso bloqueado puede estar haciendo uso de algún recurso, hay casos especiales que liberan dichos recursos.

A su vez, `kill` no puede matar a `idle` ni `shell`; el primero no puede ser asesinado de ninguna forma, mientras que el segundo puede matarse mediante el comando `exit` o la combinación de teclas `LCTRL+D` (es decir, tampoco puede matarse con `LCTRL+C`).

Si se mata al último writer o reader creado con `mvar`, se decidió matar a todos los readers o writers (respectivamente) ya que no habrá quien escriba o lea el pipe. Es decir, si se mata al último writer que se encontraba vivo, se matará también a todos los readers y se liberará el pipe.

No se implementó la posibilidad de crear semáforos anónimos, ya que los mismos no son un requerimiento y todas las funcionalidades pueden implementarse con semáforos con nombre.

Se decidió que si se bloquea a un proceso mientras se encontraba en la región crítica no se toma ninguna acción en particular, por lo que todos los procesos que utilicen ese semáforo deberán esperar a que dicho proceso se desbloquee y lo libere. Esto también sucede cuando un proceso que usa pipes es bloqueado por el usuario.

Los comandos que permiten escribir en pantalla (`cat`, `wc` y `filter`), si se ejecutan en background, se matan instantáneamente.

4. Análisis de PVS

Hay dos archivos que no corresponden a la implementación a los cuales se les indicó que PVS no los escanee, pero igualmente figuran en el reporte (`bmfs.c` y `main.c`)

Se detectan dos errores de alto nivel en `video_drivre.c`, líneas `207` y `219`, pues se modifica el valor de `i` dentro del loop y también se utiliza en su condición. Tras revisar la lógica del código, estos se consideran falsos positivos.
A su vez se detecta uno de nivel medio en la línea `344` para aclarar que un número está en octal, lo cual es correcto en la implementación, por lo que también es un falso positivo.

Se detecta un error de nivel bajo en la línea `706` de `userlib_so.c` debido a la espera activa pedida en el enunciado para la implementación de mvar.

Se detecta un error de nivel bajo en la línea `35` de `buddy_memory_manager.c` por una conversión de una dirección de memoria `void *` a `int` para manipularla matemáticamente y luego retornar otra dirección de memoria. PVS lo considera mal estilo pero consideramos que es un falso positivo.

5. Contenido del README

Instrucciones de compilación y ejecución

Requerimientos

Para la ejecución del sistema es necesario contar con la imagen de Docker de la cátedra de Sistemas Operativos, QEMU, y una copia del presente repositorio.

Imagen

Para obtener la imagen de Docker de la cátedra utilizar el comando:

```
docker pull agodio/itba-so-multi-platform:3.0
```

Inicialización del entorno de compilación

Para iniciar un contenedor temporal con la imagen descargada previamente:

```
sudo docker run --rm -v ${PWD}:/root --security-opt seccomp:unconfined --add-host=host.docker.internal:host-gateway -it agodio/itba-so-multi-platform:3.0
```

Alternativamente puede utilizarse un contenedor que cuente con dicha imagen.

Compilación

El sistema cuenta con dos administradores de memoria: uno simple ideado por los autores y otro que utiliza la técnica de alocación **buddy**. Por lo tanto, se ofrecen dos opciones de compilación, una para cada uno de los administradores.

Compilación con administrador de memoria simple

```
make all
```

Compilación con administrador de memoria *buddy*

```
make buddy
```

Eliminación de archivos compilados

```
make clean
```

Ejecución

La ejecución del sistema se realiza por fuera del entorno de compilación, sobre la terminal de un sistema operativo basado en Linux. **Es decir, no debe ejecutarse dentro del contenedor utilizado para la compilación.**

Ejecución del sistema

```
./run.sh
```

Debug y análisis de código

El sistema está preparado para ser debugueado con `gdb` y `gdb-dashboard`, además del analizador PVS-studio.

Debug

1. Compilar el sistema con las instrucciones de compilación detalladas anteriormente
2. Fuera del contenedor, ejecutar el sistema con el comando `./run.sh gdb`
3. Otra vez dentro del contenedor, ejecutar el comando `gdb`. Se iniciará `gdb-dashboard` para comenzar con el debugueo.

PVS-studio

1. Compilar el sistema con el comando `make pvs`

2. Consultar el informe generado en el directorio `/informe_completo.html/index.html`

Instrucciones de replicación

El teclado del sistema está en inglés, por lo que el mapeo de teclas coincide con este teclado y no con el de español. Se aclarará la combinación de teclas correcta en caso de ser necesario.

Comandos

Comando	Descripción	Parámetros
`exit`	Finaliza la shell	
`clear`	Limpia la pantalla	
`sleep secs`	Congela el sistema por `secs` segundos	`secs`: `int`, segundos a congelar
`help`	Imprime una lista de comandos y sus parámetros	
`registers`	Imprime el valor actual de los registros	Presionar `LALT` antes de ejecutar
`test-mm bytes`	Ejecuta el `test-mm` de la cátedra	`bytes`: `int`, bytes a pedir recurrentemente
`test-prio maxprio`	Ejecuta el `test-prio` de la cátedra	`maxprio`: `int`, máxima prioridad a aplicar
`test-pcs procs`	Ejecuta el `test-pcs` de la cátedra	`procs`: `int`, cantidad de procesos a crear
`test-sync ops sem?`	Ejecuta el `test-sync` de la cátedra	`ops`: `int`, cantidad de veces que se realizan las operaciones. `sem?`: `0|1`, `0` para no utilizar semáforos, `1` para utilizarlos
`mem`	Imprime el estado de la memoria	
`kill pid`	Mata al proceso `pid`	`pid`: `int`, identificador del proceso a matar
`ps`	Imprime el estado de todos los procesos vivos y zombie	

Comando	Descripción	Parámetros
`nice pid prio`	Cambia la prioridad del proceso `pid` a `prio`	`pid`: `int`, identificador del proceso al que modificar su prioridad.`prio`: `int`, prioridad a aplicar
`block pid`	Bloquea al proceso `pid`	`pid`: `int`, identificador del proceso a bloquear
`unblock pid`	Desbloquea al proceso `pid`	`pid`: `int`, identificador del proceso a desbloquear
`loop secs`	Imprime un mensaje en pantalla cada `secs` segundos	`secs`: `int`, segundos de espera
`wc`	Cuenta la cantidad de líneas escritas en el modo de escritura	
`cat`	Imprime en pantalla cada línea escrita en el modo de escritura	
`filter`	Imprime cada vocal de cada línea escrita en el modo de escritura	
`mvar writers readers`	Simula, mediante semáforos y un pipe, el comportamiento de una variable global sobre la que los escritores escriben una letra y los lectores la imprimen en pantalla con un color distintivo	`writers`: `int`, cantidad de escritores. `readers`: `int`, cantidad de lectores
`msg`	Imprime un mensaje en pantalla	

Caracteres especiales

Pipes `|`

Para comunicar procesos creados por comandos, utilizar el caracter `|`, mediante la combinación `LSHIFT + ` ` .

Ejemplo: `cat | filter`

Background `&`

Para ejecutar procesos creados por comandos en background en lugar de foreground, utilizar el caracter `&`, mediante la combinación `LSHIFT + 7`.

Ejemplo: `loop 1 &`

Atajos

Interrumpir ejecución `LCTRL + C`

Para interrumpir la ejecución de un proceso en foreground, utilizar `LCTRL + C`

Enviar EOF `LCTRL + D`

Para enviar EOF a un proceso en foreground, utilizar `LCTRL + D`

Ejemplos para demostrar los requerimientos

Se detallan ejemplos para requerimientos que no son triviales; en términos generales, que un requerimiento sea trivial significa que es necesario para realizar los otros ejemplos. Por ejemplo, reservar memoria es un requerimiento trivial ya que sin él no puede utilizarse la shell (entre otros problemas).

Memoria

- `mem`: imprime en pantalla la memoria total, usada y disponible.

Procesos, CS y Scheduling

- `loop 10 &`: crea un proceso en background que imprime en pantalla y renuncia al CPU cada 10 segundos. Para el resto de ejemplos se supone que este proceso es el de pid `2`.
- `ps`: imprime todos los procesos vivos o zombie junto con detalles de los mismos, entre ellos su nombre, pid, prioridad y RSP. No se indica si el mismo es foreground: consultar el informe para su justificación.
- `kill 2`: mata al proceso de pid `2`.
- `nice 2 0`: modifica la prioridad del proceso de pid `2` a la más alta, `0`.
- `block 2`: bloquea el proceso de pid `2`.
- `unblock 2`: desbloquea el proceso de pid `2`.
- `cat`: crea un proceso en foreground, hijo de la shell, que para finalizar debe recibir EOF. Una vez que finaliza vuelve a la shell. (Es decir, la shell espera a que su hijo `cat` finalice).

Sincronización

- `mvar 2 2`: crea 2 escritores y 2 lectores (del problema de sincronización de escritores y lectores) que utilizan dos semáforos (se crean, abren, cierran, bloquean y liberan).

Inter Process Communication

- `cat | wc`: crea y abre un pipe que establece el extremo de escritura de `cat` como el de lectura de `wc`, provocando que se imprima la cantidad de líneas escritas en `cat`.

Aplicaciones de User space

Consultar la sección # Comandos, utilizarlas basta para comprobar su funcionamiento.

Requerimientos faltantes o parciales

Buddy memory manager

Cuando se compila con el administrador de memoria buddy, el test del mismo (`test-mm`) rompe el sistema.

El sistema en sí funciona usando este administrador, y no se logró rastrear el error para solucionarlo.

Cerrar semáforos

Al matar un proceso que usa semáforos en su interacción con otros, si el mismo se encontraba en uso de la región crítica, no libera el uso del semáforo y los demás procesos quedan bloqueados sin manera de desbloquearlos. Además, si se matan a todos los procesos que utilizaban un semáforo, este no se cierra.

Esto sucede particularmente al utilizar `mvar`; en ejecución es imposible saber qué proceso está ejecutándose en un momento dado y si se mata a uno de ellos es posible matar al que se encontraba en la región crítica (notar además que, por diseño, aunque se mate a un writer también se bloquearán los readers, y viceversa).

A su vez, luego de matar a todos los procesos de `mvar`, los semáforos quedan abiertos.

En la rama `PipesMvar` se encuentra una solución parcial que se decidió no incorporar ya que provocaba bugs en `test-sync`.

Prioridades

El uso de prioridades presenta una complicación en la cual procesos idénticos demoran cantidades diferentes de ticks en cumplirse, provocando que no se ejecuten en el orden esperado. Los procesos guardan datos como `total_ticks` y `changes` para documentar cuantos ticks tuvieron y cuántas veces cedieron su lugar en el scheduler, mediante los cuales se verificó que las prioridades y el mecanismo round-robin implementados se respetan; pero al ejecutar `mvar` o `test-prio` los resultados no corresponden con lo esperado.

Para replicar este problema, en el modo de `debug` colocar un breakpoint `b new_test_prio.c:80`, continuar, ejecutar dicho test (`test-prio`), y colocar otro en `b proc.c:create_process`. Una vez parado en éste último podrá mirar los ticks y changes hechos por los procesos, y verificar que aunque todos los `zero_to_max` son idénticos y de misma prioridad (ya que corresponden a los de la primer tanda del test), algunos tardan más ticks que los otros.

Limitaciones

- Solo están implementadas las teclas especiales del lado izquierdo del teclado (`LCTRL`, `LSHIFT`, `LALT`)
- El scrolleo de pantalla funciona pero por problemas ajenos al alcance de este trabajo práctico presenta errores donde se imprime basura en las últimas líneas en lugar de limpiarlas para continuar la escritura. Se pide utilizar el comando `clear` **antes de llenar la pantalla** para continuar con la ejecución del sistema.
- Los procesos que imprimen en pantalla de forma recurrente, como `loop` y los readers de `mvar`, lo hacen sobre la línea de comandos pero esta escritura no forma parte del buffer de comandos. Es decir, al enviar un comando, no se tendrá en cuenta lo escrito por estos procesos.

Citas

- Como nota general, se hizo uso de clientes de IA para encarar la implementación. Todo el código generado por IA fue interpretado, modificado y adaptado sobre secciones puntuales del código, por lo que no hay una cita apropiada para el mismo.
- Tests de la cátedra: https://github.com/alejoaquili/ITBA-72.11-SO/tree/dd249451d7e7133f4e653f946dbad0077ee713bc/kernel-development/tests
- https://pdos.csail.mit.edu/6.828/2019/lec/malloc.c, para la implementación del administrador de memoria buddy.