turtleOS - User Manual

Integrante	Legajo
Alejo Flores Lucey	62622
Andrés Carro Wetzel	61655
Nehuén Gabriel Llanos	62511

Descripción general

turtleOS es un pequeño sistema operativo desarrollado a partir de Pure64, en el marco de la materia Sistemas Operativos.

Es posible interactuar con el sistema a través de una terminal, que permite ejecutar diversos comandos para verificar su funcionamiento.

Para su utilización es necesario tener acceso a un teclado; el mouse no es utilizado.

> !! turtleOS supone que se posee de un teclado con distrubución ANSI "United States".

Compilación y ejecución del kernel

Prerequisitos

Se necesitan tener instalados los siguientes paquetes para compilar y correr este proyecto:

- nasm
- qemu
- gcc
- make
- docker

Compilación

Por única vez, descargar la imagen de Docker:

```
docker pull agodio/itba-so:1.0
```

Luego, cada vez que se quiera compilar, ejecutar el siguiente comando en el directorio del proyecto:

```
docker run -v "${PWD}:/root" --privileged -ti --rm agodio/itba-so:1.0
```

Se iniciará el contenedor de Docker. Ahora, ejecutar los siguientes comandos para compilar el Toolchain:

```
cd root
cd Toolchain
make all
cd ..
```

Luego, se debe compilar el proyecto. Hay dos algoritmos de alocación de memoria disponibles, y se debe elegir cuál usar en tiempo de compilación.

Para compilar turtleOS con algoritmo de alocación de memoria Heap, utilizar:

```
make all
```

Para compilar turtleOS con algoritmo de alocación de memoria Buddy, utilizar:

```
make buddy
```

Sólo falta cerrar el contenedor de Docker:

```
exit
```

Análisis de código estático

El sistema operativo está preparado y pasa sin errores ni advertencias los siguientes analizadores de código estático:

- PVS-Studio
- CPPCheck

Para ejecutar dichos analizadores, correr los siguientes comandos en el directorio raíz del proyecto:

```
docker run -v "${PWD}:/root" --privileged -ti --rm agodio/itba-so:1.0
cd root
cd Toolchain
make all
cd ..
make static_analysis
```

Los resultados de PVS-Studio estarán disponibles en el archivo `report.tasks`. A su vez, los resultados de CPPCheck estarán disponibles en el archivo `cppcheck.txt`.

Ejecución

En un host que tenga QEMU instalado, correr:

```
./run.sh
```

Comandos disponibles

`block <pid>`

Programa que bloquea incondicionalmente al proceso asociado a el *Process ID* provisto.

> ⚠️ Este es un programa **built-in**. Esto significa que no es un proceso en sí y por lo tanto no puede ejecutarse en *background* ni utilizarse con *pipes*.

`cat`

Programa que imprime a salida estándar lo que reciba por entrada estándar.

El programa termina al leer `EOF (-1)`. Dicho caracter puede ser ingresado por el teclado utilizando `Ctrl + D`.

`clear`

Programa que limpia la pantalla y coloca el cursor en la esquina superior izquierda.

> ⚠️ Este es un programa **built-in**. Esto significa que no es un proceso en sí y por lo tanto no puede ejecutarse en *background* ni utilizarse con *pipes*.

`fibonacci`

Programa que imprime en pantalla los números de la serie de Fibonacci.

El programa termina cuando el próximo número de la serie no es representable.

`filter`

Programa que imprime a salida estándar lo que reciba por entrada estándar, a excepción de las vocales.

El programa termina al leer `EOF (-1)`. Dicho caracter puede ser ingresado por el teclado utilizando `Ctrl + D`.

`help`

Programa que despliega en pantalla una lista de comandos válidos para introducir, junto a una pequeña descripción del mismo.

`kill <pid>`

Programa que termina la ejecución del proceso asociado al *Process ID* provisto.

> ⚠️ Este es un programa **built-in**. Esto significa que no es un proceso en sí y por lo tanto no puede ejecutarse en *background* ni utilizarse con *pipes*.

`loop`

Programa que imprime en pantalla un saludo y "pausa" su ejecución por un tiempo determinado.

El proceso no termina, sino que entra en un ciclo infinito.

> ℹ️ A fines didácticos, este proceso realiza espera activa. Esto se permite, aunque esté mal conceptualmente, para verificar el correcto funcionamiento de los mecanismos de *scheduling* del sistema operativo.

`mem`

Programa que imprime el estado de la memoria del sistema operativo.

`nice <pid> <new_priority>`

Programa que cambia la prioridad del proceso asociado al *Process ID* provisto.

La prioridad debe ser un número entero mayor o igual a cero y menor o igual a ocho. Cabe mencionar que la prioridad más alta es cero, y la prioridad más baja es 8.

> ⚠️ Este es un programa **built-in**. Esto significa que no es un proceso en sí y por lo tanto no puede ejecutarse en *background* ni utilizarse con *pipes*.

`phylo`

Solución gráfica al problema de los filósofos.

Se permite agregar y quitar filósifos de la mesa utilizando las techas `a` y `r`, respectivamente.

El programa termina cuando se presiona la tecla `q`.

`pipes`

Programa que imprime el estado de todos los *pipes* del sistema operativo.

`primes`

Programa que imprime en pantalla los números primos a partír del 2.

El programa termina cuando el próximo numero primo no es representable.

`ps`

Programa que imprime el estado de todos los procesos del sistema operativo.

`test_mm <max_memory>`

Programa que ejecuta los testeos provistos por la cátedra para verificar el correcto funcionamiento del manejador de memoria.

El argumento recibido representa cuánta memoria será alocada.

`test_priority`

Programa que ejecuta los testeos provistos por la cátedra para verificar el correcto funcionamiento del cambio de prioridad de procesos.

`test_processes <max_processes>`

Programa que ejecuta los testeos provistos por la cátedra para verificar el correcto funcionamiento de la creación, terminación, bloqueo y desbloqueo de procesos.

`test_synchro <iterations> <use_sem>`

Programa que ejecuta los testeos provistos por la cátedra para verificar el correcto funcionamiento de la implementación de semáforos.

El primer argumento recibido representa la cantidad de veces que se modificará una variable con el fin de verificar el funcionamiento de los semáforos.

El segundo argumento recibido representa un *flag* que definirá si el testeo se corre utilizando semáforos o no.

Si el programa se ejecuta utilizando semáforos, el valor final impreso debe ser cero. En cambio, si el programa se ejecuta sin la utilización de semáforos, este valor final puede o no ser cero.

`time`

Programa que imprime en pantalla la fecha y hora actual. Dicha fecha y hora es desplegada en el huso horario GMT-3.

`turtle`

Programa que imprime a salida estándar la mascota del sistema operativo.

`unblock <pid>`

Programa que desbloquea al proceso asociado a el *Process ID* provisto.

> ⚠️ Este es un programa **built-in**. Esto significa que no es un proceso en sí y por lo tanto no puede ejecutarse en *background* ni utilizarse con *pipes*.

`sem`

Programa que imprime el estado de todos los semáforos del sistema operativo.

`wc`

Programa que imprime a salida estándar la cantidad de línesas leídas por entrada estándar.

El programa termina al leer `EOF (-1)`. Dicho caracter puede ser ingresado por el teclado utilizando `Ctrl + D`.

Piping

Todos los programas anteriormente descriptos que no son built-in pueden conectarse utilizando la siguiente sintáxis: `<programa_1> | <programa_2>`.

Cuando se corra dicho comando, la salida estándar del `<programa_1>` será redirigida a la entrada estándar del `<programa_2>`.

Ejecución de programas en *background*

Todos los programas anteriormente descriptos que no son built-in pueden ejecutarse en segundo plano utlizando la siguiente sintáxis: `&<programa>`.

System Calls implementadas

Se debe generar una interrupción del tipo 80 para ejecutar la *system call* deseada.

Los registros que se detallan a continuación deben poseer los siguientes parámetros para la ejecución correcta de la system call.

En *RAX* se indica qué system call se desea ejecutar.

El valor de retorno de la system call se obtendrá en *RAX*.

System Call	RAX	RDI	RSI	RDX
read	0x00	`unsigned int fd`	`char * buf`	`uint64_t count`
write	0x01	`unsigned int fd`	`char * buf`	`uint64_t count`
exec	0x02	`uint64_t program`	`unsigned int argc`	`char * argv[]`
exit	0x03	`int ret_value`	`char autokill`	
getpid	0x04			
waitpid	0x05	`pid_t pid`		
yield	0x06			
block	0x07	`pid_t pid`		
unblock	0x08	`pid_t pid`		
kill	0x09	`pid_t pid`		
nice	0x0A	`pid_t pid`	`int new_priority`	
malloc	0x0B	`uint64_t size`		
free	0x0C	`uint64_t ptr`		
sem_open	0x0D	`char * name`	`int initial_value`	
sem_close	0x0E	`sem_t sem`		
sem_wait	0x0F	`sem_t sem`		
sem_post	0x10	`sem_t sem`		
pipe	0x11	`int fds[]`		
dup2	0x12	`int old`	`int new`	
close	0x13	`int fd`		
mem_info	0x14			
pipe_info	0x15			
sem_info	0x16			

System Call	RAX	RDI	RSI	RDX
process_info	0x17			
time	0x18	`date_s * s`		
clear	0x19			

La *system call* `pipe()` recibe un vector de enteros, donde dejará los *file descriptors* asignados a la terminal de lectura y terminal de escritura del nuevo *pipe.* En `fds[0]` se almacenará el *file descriptor* correspondiente a la terminal de lectura y en `fds[1]` se almacenará el *file descriptor* correspondiente a la terminal de escritura.

Las *system calls* `mem_info()`, `pipe_info()`, `sem_info()` y `process_info()` devuleve un puntero a una estructura con los datos solicitados. El usuario es el encargado de liberar dicha memoria.