72.11 - Sistemas Operativos

Instituto Tecnológico de Buenos Aires

Trabajo Práctico Nro. 2

Construcción del Núcleo de un
Sistema Operativo y estructuras de administración de recursos.

Profesores

Godio, Ariel

Aquili, Alejo Ezequiel

Mogni, Guido Matías

Merovich, Horacio Víctor

Alumnos

Bartellini Huapalla, Mateo (61438)

Estevez, Franco Nicolas (61452)

Vittor, Lucas Agustin (61435)

Instrucciones de compilación y ejecución

Para compilar el proyecto con docker, desde la carpeta root:

```
user@user:~/kernel-pure64$ alias dcom='docker run -v "${PWD}:/root" -ti -w /root
agodio/itba-so:1.0 ${1}'
user@user:~/kernel-pure64$ cd Toolchain
user@user:~/kernel-pure64/Toolchain$ dcom make && cd ..
user@user:~/kernel-pure64$ dcom make # si queremos compilar sin buddy
user@user:~/kernel-pure64$ dcom make MM=BUDDY # si queremos compilar con buddy
```

Para ejecutar el proyecto:

```
user@user:~/kernel-pure64$ ./run.sh # en linux
user@user:~/kernel-pure64$ ./run.bat # en Windows
```

Decisiones tomadas durante el desarrollo

Physical Memory Management

Tenemos un único header para el *Buddy* y el *Heap 4* en **Kernel/include/memoryManager.h**, del que se pueden usar los siguientes métodos:

```
void heap_init(void);
void *alloc(size_t size);
void free(void *address);
void dump_mem(void);
```

Luego, existen dos archivos fuente para cada implementación del memory manager en **Kernel/memoryManager.c** y **Kernel/buddy.c** que contienen los mismos métodos del header, y que son utilizados dependiendo de cómo se haya compilado con el *Makefile*.

Para la realización del *Buddy* decidimos basarnos en la implementación de *Broken Thorn*, pero haciendo nuestra propia implementación de una *doubly linked list*. Además de esto, decidimos no hacer métodos recursivos para evitar sobrepasar el stack provisto, por lo que hicimos su adaptación a métodos iterativos.

Procesos, Context Switching y Scheduling

Priority-based round Robin

El manejo de procesos en el Kernel se encuentra en **Kernel/include/sched.h** y **Kernel/sched.c.**

En la implementación del scheduling, optamos por el uso de un arreglo denominado *processes* que contiene en cada posición la estructura *processControlBlock*.

Al iniciar el scheduler, ponemos siempre en la primera posición el PID del proceso halt y en la segunda posición el proceso que se pasa por parámetro en la función *initScheduler*.

Para la prioridad, decidimos seguir la cantidad de niveles que Linux utiliza, pero utilizando únicamente números positivos (Linux utiliza de -20 a 19 y nosotros de 0 a 39). Cada proceso creado comienza con el mínimo nivel de prioridad, que puede ser modificado luego por el usuario mediante una syscall. Manejamos un sistema de *tickets*, los cuales indican cuántos *quantums* seguidos el scheduler elegirá a este proceso para su ejecución (un proceso con prioridad 0 ejecutara 1 *quantum* y luego el scheduler elegirá el próximo listo, pero en cambio, uno con prioridad 39 ejecutará 40 *quantums* seguidos antes de que el scheduler elija otro proceso).

A continuación, mostramos cada una de las syscalls implementadas:

- **Crear y finalizar procesos:** sys_create_process, sys_kill
- **Obtener el ID del proceso que llama:** sys_get_curr_pid
- **Listar procesos:** sys_dump_processes
- **Matar a un proceso:** sys_kill
- **Modificar la prioridad de un proceso:** sys_nice
- **Cambiar el estado de un proceso:** sys_block
- **Renunciar al CPU:** sys_yield_process

Sincronización

El manejo de la implementación de sincronización en el kernel se encuentra en los archivos **Kernel/include/sem.h** y **Kernel/sem.c**

A continuación, mostramos cada una de las syscalls implementadas:

- **Crear, abrir y cerrar un semáforo:** sys_open_sem**,** sys_close_sem
- **Modificar el valor de un semáforo:** sys_post_sem, sys_wait_sem
- **Listar el estado de todos los semáforos del sistema:** sys_print_sem

La syscall de apertura, crea el semáforo si no existe ya uno con el id provisto, y devuelve un valor especial en el caso de que ya existiera. Una de las decisiones tomadas fue que al momento de cerrar el semáforo, se liberaran todos los procesos bloqueados en el mismo. Para garantizar la atomicidad de ciertas instrucciones, se utilizó la instrucción xchg.

Inter Process Communication

El manejo del Inter Process Communication se encuentra en los archivos **Kernel/include/pipes.h** y **Kernel/include/pipes.c.**

Optamos por crear una estructura que contenga la información necesaria de un pipe:

```
typedef struct pipe_t {
```

```
        fd_t fdin;
        fd_t fdout;
        int buffer[MAX_BUFFER];
        uint16_t read_pos;
        uint16_t write_pos;
        char write_sem[SEM_ID_SIZE], read_sem[SEM_ID_SIZE];
} pipe_t;
```

Podemos ver que tenemos ambos file descriptors, junto con el buffer por el cual los procesos que se conecten a este podrán leer y escribir en él, junto con las ubicaciones que indican cual es la próxima posición a leer/escribir. Por último, como esta zona podría ser accedida por más de un proceso a la vez, para evitar conflictos también decidimos incluir un semáforo para cada acción (sincronizar escribir y leer).

- **Crear, abrir y cerrar pipes:** `sys_open_pipe, sys_close_pipe`
- **Leer y escribir de un pipe:** `sys_write_pipe, sys_read_pipe`
- **Listar el estado de todos los pipes del sistema:** `sys_dump_pipes`

Comandos

- **help**
- **mem**
- **ps**
- **kill**
- **nice**
- **block**
- **sem**
- **cat**
- **wc**
- **filter**
- **pipe**
- **phylo**

Pasos a seguir para demostrar el funcionamiento de cada uno de los requerimientos

En un principio, las aplicaciones de testeo provistas por la cátedra pueden ser ejecutadas desde la terminal con sus respectivos nombres. La mayoría de estas se ejecutan sin un fin específico, por lo que se recomienda correrlas en *background*. Además, recomendamos correr los siguientes comandos:

- cat|filter
- cat|wc
- loop|filter&

Con estos comandos se puede verificar el correcto funcionamiento de los pipes y de la creación de procesos.

Limitaciones

Actualmente, varias de las implementaciones del Kernel utilizan un arreglo de tamaño fijo, lo cual se decidió por simplicidad y para evitar encontrarnos con errores durante el desarrollo de los memory managers. Se consideró la opción de usar vectores dinámicos pero se optó por no hacerlo por simplicidad. Esto lleva a que las cantidades de procesos, pipes y semáforos tengan un máximo determinado.

Problemas encontrados durante el desarrollo y cómo se solucionaron

Uno de los problemas principales que tuvimos en el desarrollo fue la planificación que nosotros hicimos de lo que tardaríamos en desarrollarlo. Al principio estábamos haciendo pair-programming para garantizar una mejor calidad del código, pero esto nos atrasó bastante. Decidimos optar por que cada uno trabaje en ramas diferentes con features específicas a desarrollar y luego sí gestionar los merges de las ramas conjuntamente para solucionar los conflictos posibles.

Luego, tomando en consideración lo aclarado por acerca del trabajo práctico pasado, decidimos sacrificar el deadline para enfocarnos en mejorar la calidad del código que presentamos, siguiendo los estándares que sigue linux. Es así que tenemos el mismo formater que linux utiliza.

Citas de fragmentos de código reutilizados de otras fuentes

Memory Manager - Heap4
FreeRTOS Kernel V10.4.3 LTS Patch 1
https://github.com/FreeRTOS

Memory Manager - Buddy
http://brokenthorn.com/Resources/OSDev26.html

Dining Philosophers problem
https://medium.com/swlh/the-dining-philosophers-problem-solution-in-c-90e2593f64e8

Linux c-lang formatter config file
https://github.com/torvalds/linux/blob/master/.clang-format

Modificaciones realizadas a las aplicaciones de test provistas

Para todas las aplicaciones de test, se agregaron las *syscalls* necesarias dentro de los *handlers* provistos. En ciertas ocasiones, como una operación podía proveer resultados de éxito, fallo y otros, se agregaron validaciones adicionales en los ifs. Se resolvieron los *TODO* que se encontraban presentes en el *test* de *Memory Management*.