Instituto Tecnológico de Buenos Aires

Trabajo Práctico N° 2

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

Perez de Gracia, Mateo
Quian Blanco, Francisco
Stanfield, Theo

Sistemas Operativos - 72.11

Segundo cuatrimestre 2023 - Grupo 7

Tabla de contenidos

1 Introducción

A lo largo de nuestro recorrido en la materia, hemos explorado el manejo de la API de sistemas operativos UNIX, adquiriendo valiosas destrezas en el proceso. Ahora, nos embarcamos en la tarea de crear nuestro propio kernel, basándonos en el proyecto final de la asignatura Arquitectura de Computadoras. Este desafío implica la implementación integral de aspectos clave, como la administración de memoria, la gestión de procesos, la planificación (scheduling) eficiente, así como la integración de mecanismos de Comunicación entre Procesos (IPC) y estrategias de sincronización.

2 Instrucciones de compilación y ejecución

El comando `./compile.sh` realizará las siguientes tareas:

- Descargará la imágen de `agodio/itba-so:2.0` de docker donde se compilará el proyecto.
- Creará un nuevo contenedor preparado para compilar el proyecto como si fuéramos nosotros porque crea un usuario con nuestro mismo **uid** y **gid**.
- Iniciará el contenedor para que siempre esté corriendo.
- Enviará los comandos necesarios para limpiar y compilar el proyecto completo.

Esa es la funcionalidad base del comando `./compile.sh`. A su vez, cuenta con dos flags que pueden ser enviados:

- `-d`: Compila el proyecto con información de debugging para poder utilizar gdb y ejecutar paso a paso el código.
- `-r`: Una vez compilado el proyecto completo, si no ocurrió ningún error durante el proceso, se ejecuta el comando `./run.sh`, es decir, una vez compilado el proyecto, lo ejecuta.

  Nota: Si se corre el comando con ambos flags en simultáneo, es decir, `./compile.sh -d -r`, el proyecto se compilará y ejecutará en modo debugging. Si solo se utiliza el flag `-r`, se compilará en modo normal y se ejecutará en modo normal, sin información de debugging.

Para cambiar el `memory manager` utilizado, debe ir a el archivo `Makefile.inc` dentro de la carpeta `Kernel` y debajo del todo podra observar el comando `MEMORY_MANAGER`. Utilice la primer opcion para elegir el memory manager de listas y el segundo para utilizar el memory manager buddy

```
MEMORY_MANAGER=USE_LIST
MEMORY_MANAGER=USE_BUDDY
```

3 Pasos a seguir

Una vez ejecutado el comando `./compile.sh` se abrirá la terminal y se deberá ejecutar el comando `help` para poder apreciar todos los distintos comandos que ofrecemos, los cuales son:

3.1 Comandos

3.1.1 Comandos generales

- `help`: muestra una lista con todos los comandos disponibles.
- `datetime`: Imprime la fecha y hora del momento.
- `setcolor`: Coloca los colores de la terminal. Como primer argumento recibe el componente a cambiar y como segundo argumento el color elegido (mensaje de ayuda al ejecutar sin argumentos).
- `switchcolors`: Invierte los colores del fondo de pantalla y del texto.
- `clear`: Limpia la pantalla.

3.1.2 Comandos de memoria

- `mem`: Imprime el estado de la memoria.

3.1.3 Comandos de procesos

- `ps`: Imprime la lista de todos los procesos con sus propiedades: nombre, ID, prioridad, stack y base pointer, foreground.
- `loop`: Imprime su ID con un saludo cada una determinada cantidad de segundos.
- `kill`: Mata un proceso dado su ID.
- `nice`: Cambia la prioridad de un proceso dado su ID y la nueva prioridad.
- `block`: Cambia el estado de un proceso entre bloqueado y listo dado su ID

3.1.4 Comandos de IPC

- `cat`: Imprime el stdin tal como lo recibe.
- `wc`: Cuenta la cantidad de líneas del input.
- `filter`: Filtra las vocales del input.
- `phylo`: Implementa el problema de los filósofos comensales. Recibe un argumento el cual indica la cantidad de filósofos en la mesa.

3.1.5 Tests

- `testioe`: Genera una `Invalid Op Code Error`.
- `testzde`: Genera una `Zero Division Erro`.
- `testmm`: Aloca memoria y la desaloca imprimiendo el estado de la misma en el proceso.
- `testproc`: Crea un proceso, que crea y elimina múltiples procesos hijos hasta que se pare el proceso. Recibe la cantidad de procesos hijos que puede crear en simultaneo.
- `testprio`: Genera tres procesos y le cambia la prioridad, imprime el id del proceso.
- `testsync`: Crea pares de procesos, los cuales aumentan y disminuyen el valor de una variable. Su primer argumento es la cantidad de operaciones que realiza cada proceso (ya se aumentar la variable en 1 o disminuirla en 1) y su segunda parámetro es si se desea usar semáforo o no (0 para no utilizar semáforo y 1 para utilizar semáforo).
- `testpipes`: Crea dos procesos y los conecta mediante un pipe para poder escribir y leer a través de él.

4 Decisiones tomadas

4.1 Estructura

En cuanto a la estructura de los archivos en Kernel, utilizamos la estructura que ya habíamos definido en Arquitectura de Computadores. Decidimos crear una carpeta llamada ipc para los mecanismos de comunicación, una carpeta de processes para los procesos y el scheduler y por último decidimos incluir los archivos de memoria en la carpeta denominada drivers.

Mirando Userland creamos la carpeta tests para incluir los tests provistos por la cátedra y la carpeta programas para incluir todos los programas creados.

4.2 Código

A la hora de crear un proceso, para guardar los files descriptors, el contexto del proceso, y el estado del proceso (para el scheduler), decidimos usar distintos structs con la intención de agrupar datos que se asignan a un proceso en específico.

```
typedef struct
{
    int8_t priority;
    uint8_t exit_status;
    ProcessStatus status;
    void* rsp;
} ProcessState;

typedef struct
{
    ReadCallback read_callback;
    WriteCallback write_callback;
    CloseCallback close_callback;
    DupCallback dup_callback;
} FileDescriptor;

typedef struct
{
    void *stack_end, *stack_start;
    uint8_t is_fg;
    char* name;

    int argc;
    char** argv;

    FileDescriptor fds[MAX_FDS];
    Queue waiting_pids;

    int parent_pid;
    Queue child_pids;

    KillCallback kill_callback;
} ProcessContext;
```

Otra decisión tomada fue la de crear una `queue` de ID de procesos. Al tener que juntar procesos en distintas partes del código, decidimos realizar una cola de `uint8_t` para poder acceder a esta cola y poder agregar, eliminar u bloquear procesos adjuntos.

```
struct node
{
    uint8_t elem;
    struct node* next;
};

struct queue_adt
{
    struct node* first;
    struct node* last;
    int count;
};
```

4.3 Código estático

Para ahorrar mucho trabajo y como bien menciono la catedra decidimos hacer las cosas estáticas, es decir, hay una cantidad de procesos máximos que se pueden tener creados al mismo tiempo.

```
static ProcessContext processes[MAX_PROCESSES];
```

Aplicamos este mismo razonamiento para otras implementaciones, por ejemplo, para pipes y semáforos, los cuales existen una máxima cantidad que se pueden crear, lo cual lleva a una de las limitaciones de nuestro código.

```
static Pipe* pipe_table[MAX_PIPES];
static SemInfo* semaphores[MAX_SEMAPHORES];
```

5 Limitaciones

5.1 Código estático

Como se mencionó en las decisiones tomadas, una gran limitación del código que se deicidio escribir, es la elección de arrays estáticos. Al tomar esta decisión hay una máxima cantidad de procesos que el usuario puede crear, el cual es la principal limitación del código. También hay una máxima cantidad de semáforos y pipes que se pueden crear, lo cual añaden a las limitaciones de nuestro código.

6 Problemas encontrados

6.1 `CTRL+C` en dilema de filósofos

Al abortar el programa de filósofos a la mitad, no se realiza el free de los semáforos y no se los elimina del array estático. Por lo cual en cualquier programa que utilice muchos semáforos y se cancele al medio, ocupara mucho espacio en los semáforos. Lo que llevara a que no se puedan crear más en cierto momento. Si se ejecutaba el programa `philo` y se lo aborta varias veces, se obtendrá un error que indica que no hay más espacio para crear semáforos.

La solución planteada es el agregado de un `kill callback`, el cual es un puntero a función que se debe correr cada vez que se mata a un proceso. En caso de que se asigne una función `kill callback` esta se llama cuando el proceso muere.

```
if (process->kill_callback != NULL)
        process->kill_callback();
```

Por eso, si se interrumpe un proceso el cual aloco memoria, este proceso llama a una función que haga el free de la memoria. En el caso de los filósofos, esta función realiza el free de los semáforos y de los nombres de los filósofos para que la función `philo` pueda ser llamada nuevamente.

6.2 Creado de queue

Un problema grande que tuvimos fue con el creado del queue. Al crear el archivo por primera vez, creamos muchas funciones que podíamos llegar a usar. El problema surgió al testear la memoria utilizando el comando `mem`. Podíamos observar con ambos memory manager un `memory leak` de distintos tamaños (para ambos memory manager) cada vez que se creaba un proceso. Por esta razón, estuvimos mucho tiempo debbugeando hasta concluir que la liberación de memoria de las `queues` estaba mal.

Si se ejecutan dos `mem` seguidos se puede ver como aumenta la memoria, lo cual podría parecer un memory leak, pero esto es incorrecto ya que si seguimos ejecutándolo deja de aumentar la memoria usada. Esto es debido a la creación de las `queues`, especialmente la `queue` que guarda los id de los proceses hijos. En el creado del proceso no se inicializa la `queue` de los procesos hijos, sino que es el primer hijo el cual crea esta `queue` a su padre. Se decidió realizarlo de esta manera porque muchos procesos no tienen proceso hijos, por lo cual asignar ese espacio de memoria en la creación de un proceso nos pareció innecesario. Por ende, el aumento en memoria que aparece en los primeros dos `mem` es esta `queue` siendo creada por primera vez, por su primer hijo.

6.3 Al matar un proceso, matar a los hijos

Un problema que encontramos al final del trabajo fue con las `syscall kill`. Al matar un proceso, ya sea a su finalización o con el `CTRL+C` mientras está corriendo, si el proceso tiene hijos estos también deber morir, ya que no serán utilizados. Por esta razón decidimos incluir en el `struct` del proceso la cola `Queue child_pids` y el entero `parent_pid`, con propósito de poder almacenar los hijos y el pid del padre para que, al matar el proceso, también podamos matar a los hijos.

Por lo tanto, una vez que decidimos matar un proceso aparte de eliminar este mismo proceso también se corre el siguiente código:

```
int child_pid;
while ((child_pid = queue_pop(process->child_pids)) != -1)
    proc_kill(child_pid, -1);
mm_free(process->child_pids);

ProcessContext* parent_proc;
if (get_process_from_pid(process->parent_pid, &parent_proc))
    queue_remove(parent_proc->child_pids, pid);
```

El `while` agarra un proceso hijo a la vez y lo mata, mientras que el `if` pregunta si este proceso tiene un padre, y se lo tienen se elimina de la lista del padre.

Para poner un ejemplo, si se ejecuta `testproc` el cual crea muchos procesos, al eliminar este proceso, también se eliminan todos los procesos creados por el `testproc` y este mismo se elimina de la `queue` de su padre, el cual vendría a ser la `shell`.

7 Modificaciones realizadas a tests provistos

Las principales funcionalidades de los tests no fueron cambiadas, los resultados esperados siguen siendo los mismos. Algunos cambios fueron:

- Los `printf` fueron mayormente reemplazados por `putchar` o `puts` según el caso de uso
- Junto con la inclusión del `puts`, incluimos nuestros colores para poder imprimir por pantalla con los colores correctos, por ejemplo, imprimir los errores de color rojo.

- Las syscalls denominadas `my_sys` en el archivo `syscalls.c` fueron adaptadas a nuestro código, haciendo la llamada a assembler correspondiente y retornando el valor correcto.
- Creamos un test para comprobar el buen funcionamiento de los pipes.

8 Citas de fragmentos de código reutilizados

Para las syscalls nos basamos en el código de las syscalls de Linux. Esta lógica fue útil exceptuando para las syscalls de semáforos ya que Linux no tiene ID para estas, por lo cual decidimos poner estas syscalls a partir del ID número 50.