Decisiones de diseño

Scheduling

El algoritmo de scheduling utilizado es round robin con prioridad, tal como solicito la cátedra, mediante el uso de una linked list donde se guarda la información de todos los procesos, incluyendo su RSP, estado, PPID, PID, file descriptors, prioridad, tiempo restante de CPU, el pointer a su stack de memoria, sus argumentos y una lista de los procesos esperando a que finalice.

Prioridad

Se definió como un número del 1 al 10, que determina durante cuantas interrupciones de timer tick consecutivas se ejecutara el proceso.

Estados

Se definieron tres estados, **READY**, **BLOCKED** y **KILLED**. Los procesos ready y blocked se encuentran todos en la lista de procesos, mientras que apenas cambia el estado de un proceso a killed este es removido de la lista, se cierran sus file descriptors, se despiertan todos los procesos que esperaban su final, y se libera la memoria utilizada para sus argumentos, stack y nodo.

File descriptors

Dado que solo tenemos pipes, los file descriptors realmente guardan un puntero a la memoria donde está guardado el pipe, junto a un flag que determina si es el extremo de escritura. Dado que el keyboard y la

consola no utilizan el sistema de pipes para enviar su información, los punteros son inicializados con el valor `NULL`, y en caso de tratar de escribir o leer a los file descriptors designados como `STDIN`, `STDOUT` o `STDERR` se fija si se sobreescribió un pipe a estos, y en caso de no hacerlo redirige la llamada a la función adecuada.

Los file descriptors son heredados del proceso padre cuando un proceso nuevo es creado, lo que permite que la consola redirija la salida de los comandos llamados de forma transparente para esos utilizando una syscall basada en el dup2 de Linux.

Administración de memoria

Dumb Manager

El administrador de memoria elegido para implementar aparte del Buddy, que se decidió nombrar Dumb por su simpleza e ineficiencia, fue basado en el del libro "The C Programming Language" de Kernighan y Ritchie, con algunas modificaciones: en vez de pedir memoria al sistema operativo, utiliza un pedazo fijo determinado por el kernel al inicializar, definido en la posición `0x600000` y con un tamaño de 128 MB, y también se implementó un sistema para tomar cuenta de los punteros otorgados, asi el hacer free de memoria que no fue otorgada no genera problemas.

Buddy

El administrador Buddy es una biblioteca creada por Stanislav Paskalev, donde se eliminó todo el código de debug, dado que hacía uso de la función printf de la librería estándar de C que no fue implementada, junto a problemas en tiempo de compilación por el retorno de floats.

Pipes

El diseño de las pipes fue basado en el de XV&. Se implementó un mecanismo para crear named pipes que difiere de las unnamed en su creación, pero funcionalmente trabajan igual.

También, se reemplazó el uso de canales para despertar procesos de XV6 por una linked list donde se almacenan los PID de los procesos esperando a leer o escribir del pipe.

Semáforos

Utilizando una linked list, se tienen todos los semáforos del sistema en un lugar. Esta lista esta protegida por un lock y el uso de spinlocks, al igual que cada semáforo individualmente. Cuando un proceso hace wait de un semáforo con valor 0, se agrega al final de una lista de PID del semáforo, y cuando se hace post se despierta al primer proceso de la lista que no haya terminado.

Keyboard

El teclado es casi el mismo que el heredado del trabajo práctico de Arquitectura de Computadoras, pero ahora hace uso de un semáforo para controlar la cantidad de caracteres disponibles para la lectura. Así, cuando un proceso lee del teclado, actúa de forma similar a un pipe y es bloqueado hasta que recibe todos los caracteres que solicito. Tambien se añadio el uso de la combinación de teclas Ctrl+D, que envía al buffer el caracter ASCII `4`, que es interpretado como `EOF` por los procesos leyendo de este.

Instrucciones

Compilación

Requiere del uso de docker, la imagen de docker compartido por la cátedra y qemu. Para instalar docker y qemu, se recomienda seguir las instrucciones en sus respectivas páginas, mientras que para instalar la imagen:

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

Luego, estando en la carpeta principal del proyecto, correr el contenedor mediante el comando

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

Y una vez en la imagen de docker, correr el siguiente comando para moverse a la carpeta donde está almacenado el proyecto:

```
cd ~
```

Para compilar, se hace uso de Make, con el uso de una variable para distinguir si se desea compilar con el administrador de memoria Dumb o Buddy. Primero se debe compilar el toolchain, mediante los siguientes comandos:

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

Una vez compilado el toolchain, se puede compilar el proyecto en sí mediante el uso del siguiente comando

```
make MM=[BUDDY | DUMB] [target]
```

El make tiene cuatro targets útiles para su compilación:

- `clean`: limpia todos los archivos compilados por el resto de comandos de este makefile.
- `all`: Compila todo el proyecto.
- `gdb`: Compila todo el proyecto con información para debugging.
- `pvs`: Corre el analizador estático de código PVS-Studio. Antes de usarlo, se debe instalar PVS y registrar una licencia en la imagen de docker.

Ejecución

Para ejecutar el proyecto, se recomienda utilizar el shell script `run.sh` en la carpeta principal del proyecto. Este acepta un argumento, que en caso de ser `gdb`, ejecutara el proyecto en qemu de forma que solo se

sigan las instrucciones del debugger.

Para debuggear, recomendamos el uso de gdb dentro de la imagen de Docker, haciendo uso del `.gdbinit` que se encuentra en la carpeta principal del repositorio.

Probablemente, el script necesite permisos adicionales, en ese caso utilizar el siguiente comando:

```
sudo ./run.sh [gdb]
```

Pasos a seguir

Shell

La shell acepta comandos, finalizándolos con un enter, y tienen una cantidad fija de argumentos. Esto quiere decir que si un comando necesita tres argumentos y recibe dos, el shell imprimirá error y volverá a solicitar input.

Para correr procesos en background se debe poner el carácter '&' al final del comando, separado por un espacio del resto de argumentos por al menos un espacio.

Para utilizar pipes no se pueden poner en background, solo se pueden encadenar dos comandos, de la siguiente forma:

```
comando1 [argumentos] | comando2 [argumentos]
```

En caso de no encontrar que tecla imprime un carácter, en el archivo `Kernel/keyboardDriver.c` están definidos los arrays que mapean cada tecla del teclado a los caracteres escritos en el buffer.

Comandos disponibles

- `help`: muestra los comandos disponibles, junto a una breve descripción de lo que hacen.
- `mem`: Imprime la memoria total, usada y libre a pantalla.
- `ps`: Imprime la información de todos los procesos bloqueados o corriendo, incluyendo su nombre, prioridad, estado, PID, PPID, base pointer y stack pointer.
- `loop`: Imprime su PID junto a un saludo cada cierto tiempo. Nunca termina.
- `kill [PID]`: Recibe un PID por argumento, y mata al proceso seleccionado. En caso de ser la shell, el resto de procesos siguen ejecutando y al terminar se queda en un loop infinito de `hlt`.
- `nice [PID] [PRIORIDAD]`: Cambia la prioridad del proceso seleccionado, la prioridad debe ser un número entero entre 1 y 10.
- `block [PID]`: Si el proceso esta corriendo lo bloquea, y si ya se encontraba bloqueado lo desbloquea.
- `sem`: Imprime a pantalla la información de los semáforos abiertos en el sistema: nombre, id, valor, número de procesos esperándolo.
- `cat`: Imprime a STDOUT lo que recibe por `STDIN`. Leyendo del teclado, termina al recibir Ctrl+D.

- `wc`: Cuanta la cantidad de `\n` leídos de `STDIN`, imprime al final la cantidad. Leyendo del teclado, termina al recibir Ctrl+D.
- `filter`: Lee de `STDIN` y lo imprime a `STDOUT`, excepto las vocales. Leyendo del teclado, termina al recibir Ctrl+D.
- `pipe`: Imprime la lista de todos los pipes: bytes leídos, escritos, cuantos procesos lo abrieron para leer, cuantos para escribir y el nombre del pipe si lo tiene.
- `phylo`: Problema de los filósofos interactivo. Con 'a' se agrega un filósofo, y con 'r' se elimina uno.
- `processtest [CANT] [CICLOS]`: Crea `CANT` procesos, los mata o bloquea y desbloquea de forma random hasta que todos hayan sido matados, `CICLOS` veces. Si `CICLOS` es 0, nunca termina.
- `prioritytest`: Demuestra el funcionamiento de las prioridades del scheduler.
- `synctest [N] [USE_SEM]`: Demuestra la sincronización de procesos, modificando una variable N veces. USE_SEM debe ser un entero positivo o cero, si es 0 no hace uso de semáforos.
- `memtest [BYTES] [CICLOS]`: Testea el administrador de memoria pidiendo memoria, escribiendo y comprobando que no se superponen. Si `BYTES` es mayor que la memoria libre disponible, se limita a la memoria libre. Si `CICLOS` es 0, nunca termina.

Limitaciones

El scheduler no guarda los valores de retorno de funciones que terminaron, por lo que `waitpid` de un proceso no devuelve su valor de retorno.

Por el formato de file descriptors que formamos, no se puede prohibir que un proceso imprima a `STDOUT` o lea de `STDIN`.

Como en los procesos no se guarda la información de que semáforos tienen abiertos, al matar un proceso con un semáforo abierto este no es cerrado.

Problemas encontrados

En un momento, como el wildcard `*` del makefile de Userland no tiene forma de ordenar los archivos, paso que al realizar el salto a la posición de memoria `0x400000` en vez de iniciar la función loader de Userland, terminaba en una función cualquiera, pero siempre la misma. Se identificó rápidamente que se trataba de un problema con el linkediteo, y para resolverlo se separó el código de Userland en sub carpetas, dejando a `_loader.c` en una posición distinguida para que siempre se encuentre al principio en el comando de linkedición.

Al ejecutar el test de procesos, ocurría que al llegar al PID 256 empezaban a ocurrir problemas al matar los procesos. Resulta que el PID de un proceso estaba siendo almacenado en un `uint8_t`, y para resolverlo lo cambiamos por un `uint64_t`.

Al matar o terminar un proceso ocurrían problemas con la memoria al realizar el free del array de argumentos. Resulta que el error provenía de no guardar el valor de `argc` en el contexto de un proceso al crearlo, por lo que se ejecutaban frees de memoria que no existía.

Al principio, para manejar los punteros otorgados por el administrador de memoria Dumb, se decidió usar la lista de `uint64_t` que ya había sido implementada en un TAD, lo que termino siendo una horrible idea

dado que se convertía en una llamada recursiva infinita al reservar memoria para cada nodo nuevo. Por esto se pasó a un array de punteros que crece en bloques de 512 posiciones.

Al terminar un proceso, cuando se cerraban sus file descriptors y se despertaban los procesos esperándolo, el proceso era interrumpido por timer tick luego de haber sido marcado como `KILLED` pero antes de terminar su liberación y se pasaba al siguiente en la lista. Para solucionar este comportamiento, se creó una syscall especial para finalizar un proceso, asegurándose que no sea interrumpido. En caso de no tener syscalls bloqueantes, podría directamente no realizar el cambio de estado y pasar directo al cierre de file descriptors y despertado de los procesos antes de eliminar el proceso de la lista, pero de esta forma también le otorgamos a Userland la oportunidad de hacer uso de un comando exit para terminar en cualquier momento.

Código Reutilizado

La implementación de las pipes está basada en esta de XV6.

La implementación de la shell está basada en la presentada en este artículo (link archivado)

La implementación del administrador de memoria Buddy fue obtenida en su totalidad de este repositorio.

Advertencias PVS

Todas las warnings de `bmfs.c` no fueron modificadas simplemente porque es el código que ya venía de x64Barebones.

1. `naiveConsole.c`:
    - Línea 11: PVS-Studio da una advertencia que solo se está definiendo el primer elemento del array, lo cual es cierto. El problema es que si se cambia por `{0}`, lo cual dejaría todo el array en zero, o no se inicializa, al correr el proyecto se genera una excepción de Pure64. Como esto es algo que viene desde x64Barebones, se decidio dejarlo sin más investigación.
    - Línea 13 y 14: PVS-Studio nota que estamos convirtiendo una constante a un pointer, lo cual en este caso está bien, pues según el manual de Pure64 la memoria de video se encuentra en esta dirección de memoria.
2. `buddy.h`: En la línea 1252, PVS-Studio da error de que el operador `<<` tiene comportamiento sin definir. Es claro por el comentario anterior al operador, escrito por el autor del administrador de memoria, que ya está asegurado que el shift no tenga comportamiento indefinido.
3. `kernel.c`: Vuelve a notar la conversión de constantes a punteros, pero en este caso es correcto su uso, pues el script de linkediteo asegura que los módulos utilicen esas direcciones de memoria para sus direcciones.