Informe

Barça Home

Grupo: Barsa(Grupo 6)

Integrantes:

nombre:	legajo
Francisco letterio	58408
Francisco Choi	59285
Guido Barbieri	59567

Decisiones

Scheduling

Para el desarrollo del scheduler, se decidió utilizar dos estructuras para guardar la información que se quiere conservar de cada proceso. Por un lado, se encuentra la información que es propia del proceso como puede ser el PID, la prioridad o donde se encuentra el stack pointer, por otro lado, se encuentran los datos que son específicos del scheduler de los procesos que están corriendo que también contienen toda la estructura anteriormente mencionada. Esta decisión se tomó por una parte, por una cuestión de modularizar y administrar más compactamente la información y por otro lado, porque hay datos que manejan específicamente los procesos como puede ser su prioridad y otros que los debería usar solo el scheduler como puede ser cuántas veces se ejecutó el proceso antes del context switch.

Otra decisión que se tomó es que se produzca una interrupción cuando se setea el estado de un proceso(sea cual fuere) y cuando se “mata” un proceso también, ya que existía el peligro de que se bloquee o se “mate” el proceso que se estaba ejecutando. Se podría haber chequeado estos factores y solo producir la interrupción del timer tick en esta situación, pero se tomó en cuenta que en el peor de los casos se estaría desperdiciando un quantum y dada la frecuencia en que se usa estas syscalls era despreciable. Por otro lado, aunque sea ínfimamente menor, dada la situación en que constantemente se esté usando este comando o syscall, también se estaría desperdiciando tiempo en chequear el edge case mencionado, por lo cual sería casi equivalente.

Luego, se puede destacar que para hacer que un proceso corra, se debe ejecutar la syscall “create process” que devuelve un puntero a la estructura que contiene su información y luego pasarlo como argumento a la syscall “run process” con su respectivo estado.

Por último, se consideró que el proceso halt debía crearse cuando se inicia toda la estructura del scheduler, ya que es necesario correr dicha función en caso de que no haya ningún otro corriendo para no hacer busy waiting. Aunque haya problemas de vulnerabilidad, como que se cambie el

estado de un proceso como es halt se tomó en cuenta que el trabajo no está hecho como para que un usuario trate de hackear el sistema.

Sincronización

En la parte de sincronización no hubieron muchas decisiones tomadas, ya que se siguió básicamente lo que recomendaba la cátedra. En este trabajo solo nos manejamos con semáforos binarios, ya que no era lo único necesario para implementar todas las funciones del tp. Por otro lado, también se puede aclarar de la razón por la cual se utilizó un clear interrupts para proteger la variable de cantidad de referencias a un semáforo. Nos surgió el problema de que el semáforo debía guardarse cuántos procesos lo habían abierto, ya que se debe liberar la memoria cuando lo cierre el último proceso.

IPC

En la parte de IPC, nos basamos en el uso de file descriptors para acceder a una zona crítica en la memoria. Nos pareció interesante la idea de que toda la comunicación entre diferentes procesos esté dado por file descriptors y un simple almacenamiento de semáforos. Aunque, se podría decir que incumple la consigna porque son pipes bidireccionales y no se pudo restringir la lectura y la escritura en ninguno de los dos lados, aunque sí cumple de que sean bloqueantes. Por último, se puede destacar que el teclado también se maneja por un pipe y, por ende, no se hace busy waiting para recibir el input de teclado en la terminal.

Administrador de memoria

Nos basamos en la base de la implementación de first-fit y buddy como dice la consigna, los cuales se manejan en chunks de 1024 bytes y tienen un límite de aproximadamente 100000 bytes. El algoritmo de buddy se implementó a través de un árbol binario y el first-fit se tomó como si fuera un bloque de memoria que podía ser subdividido de a 1024 bytes.

Checkeo de requerimientos

Scheduler

- **Priority-Based Round Robin**: se puede checkear utilizando como background el proceso “loop” utilizando el comando (“& loop 5”), viendo como se van intercambiando los procesos a lo largo de tiempo.
- **Obtener el proceso que llama:** Se implementó una syscall que se llama “sys_get_pid”. No tenía sentido ponerlo en la terminal, ya que siempre se iba a printear el pid de la terminal.
- **Listar los procesos que están corriendo:** comando “ps”
- **Matar un proceso arbitrario:** comando “kill <pid>”
- **Modificar la prioridad de un proceso arbitrario:** comando “nice <pid> <prioridad>”
- **Modificar el estado de un proceso arbitrario:** comando “block/unblock <pid>”

Memory manager

- **Reservar memoria:** función “giveMeMemory”
- **Liberar memoria:** función “unGiveMeMemory”
- **Cambiar el algoritmo de reserva de memoria en tiempo de compilación:**
  En el makefile de kernel cambiar la siguiente línea

```
SOURCES=$(wildcard *.c) $(wildcard mmu/memoryManager.c) $(wildcard
scheduler/*.c)
```

  En donde dice “memoryManager.c” utilizar:
  - memoryManager.c : **first fit**
  - buddy.c : **buddy**

- **Checkeo de memoria:** comando mem

Sincronización

- **Crear, abrir y cerrar un semáforo:**

Crear y abrir:

int sid = sys_create_semaphore(<nombre>, <estado>);
//el estado es SEM_LOCKED o SEM_UNLOCKED y el nombre es un int, ya
//que era más fácil para el checkeo

Cerrar:
sys_sem_close(sem_id);

- **Modificar el valor de un semáforo:** sys_sem_wait(<sem_id>) o sys_sem_post(<sem_id>)
- **Listar los estados de los semáforos:** comando “sem”
- **Modificar el valor de un semáforo:** sys_sem_wait

IPC

- **Listar el estado de todos los pipes:** comando “pipe”
- **Conectar el stdin de un programa con el stdout de otro:** Para probar esta funcionalidad se puede usar los comandos: “cat”, “filter” y “wc” en las cuales se utiliza el stdin y el stdout para recibir y devolver los resultados. Ejemplo: “cat hola | filter”
- **Crear y abrir pipes: “**sys_open_pipe” para abrir y “sys_close_pipe” para cerrar

Aplicaciones de User Space

Todas las funciones están disponibles como fueron solicitadas. Las funciones y los parámetros tienen que estar separadas por un espacio(ejemplo: “nice 3 1” o “block 4”). Luego, para ejecutar comandos en background el “&” va luego del comando a ejecutar(ejemplo: “& loop 4”). Luego, el pipe se usa exactamente igual que en la terminal de linux. Por último, el comando phylo capta tres posibles opciones del teclado:

- “a” para agregar un filósofo
- “s” para sacar un filósofo
- “q” para salir del programa

Limitaciones

Hay varias limitaciones en el uso de la aplicación “phylo” se observó que hay veces que la terminal se queda bloqueada cuando recibe por entrada del teclado dos veces la adición de un filósofo seguidas. Se sospecha que esto ocurre por una desincronización entre todos las funciones que intervienen en esa aplicación pero no se pudo “fixear” a tiempo. Por otro lado, también hay varias limitaciones con el pipe, ya que hay varios casos en que si se ejecuta en la terminal no responde como debería, esto ocurre por errores en el parseo y por querer poder recibir parámetros por argumento y no limitarlo a sólo el uso del stdin, ya que nos parecía la manera más fácil y útil de comprobar el funcionamiento de dicha aplicación. Por último, se observó que hay un error que se arrastra desde el TP de Arquitectura de Computadoras del driver de teclado por lo que no se pueden tipear varias letras rápido, porque no detecta las interrupciones en dicho caso. Hay muchos warnings del trabajo práctico de arquitectura que sabíamos que eran incorrectos, pero no llegamos a limpiar todos, al igual que warnings de “implicit declaration” que no sabemos de dónde salen, ya que las carpetas están incluídas en el archivo.

Problemas

- Tuvimos que rehacer la terminal porque la terminal del trabajo práctico de arquitectura sólo permitía la ejecución de aplicaciones del tipo (void * (void)), lo cual nos demoró bastante tiempo.
  Solución: Rehacer la terminal a partir de la división por “tokens”
- La actualización del sistema operativo de mac impidió el avance del proyecto por varios días dado que el scheduler no pudo ser avanzado.
  Solución: Descarga de virtual machine y pushear y pullear de un repositorio git.
- La poca experiencia debuggeando programas de bajo nivel demoraron la búsqueda de errores que eran mínimos pero que a nivel de impacto en ejecución eran muy grandes.
  Solución: Prueba y error
- Hubieron errores en el scheduler y los semáforos que retrasaron el avance en otras áreas, como por ejemplo pipes y la ejecución de las aplicaciones.

Solución: Tratar de pensar nuevamente la teoría que avala cada función y ver los posibles casos especiales en la que se está metiendo el programa.

Fragmentos de código reutilizado

1 - Generación de un número al azar: Se utilizó una generación de números al azar que nos daba una mejor distribución de números, el cual no fue utilizado dado la complejidad del algoritmo para lo que queríamos implementar.

2 - Int to String(itoa): Reutilizamos el int to string para transformar un string en un int en la aplicación wc.

3 - Aplicación filósofos: No copiamos exactamente igual a la implementación de los filósofos de Tannenbaum, ya que la tuvimos que adaptar a nuestro sistema operativo, pero el código está basado en dicha solución.

Instrucciones de compilacion y ejecucion:

Para la instalación se requiere tener instalado docker. Además se requiere tener qemu para su ejecución. Para su instalación en una computadora con ubuntu se puede utilizar los siguientes comandos

```
sudo apt install docker
sudo apt install docker.io
sudo apt install qemu
sudo apt install qemu-system
```

Una vez instalados docker y qemu se puede correr el archivo docker.sh, el mismo obtiene la imagen de docker provista por la cátedra y luego compila con la misma el programa. Por ultimo, corre el script run.sh el cual abre la imagen generada con qemu. Una vez corrido docker.sh, se puede utilizar run.sh para ejecutarlo directamente

También se provee de otros dos scripts, utilizados para debuggear. Los mismos son dockerWithDebugger.sh y runWithDebugger.sh. Estos se pueden usar para debuggear con gdb. También hay un ejemplo de los comandos que se deben correr en gdb. Por último hay un archivo(gdbMinorInterface) para ejecutar una mínima interfaz en gdb que corre usando python. Para correr el

mismo se debe correr gdb y una vez en él, desde la carpeta del proyecto, ejecutar el siguiente comando

```
source gdbMinorInterface
```