Sistemas OperativosInter Process Communication

línea corta

Fermín Gómez

Miguel Di Luca

Pedro Vedoya

Maite Herrán

Índice

[**1. Introducción**](#_kdylzzkhzs1q) **2**

[**2.Implementación**](#_blfo6enubdja) **2**

[2.1.Pipes](#_m747pjvk0f02) 2

[2.2.Manejo de Memoria](#_sdfrh6m60aw6) 2

[2.3.Scheduler](#_dpm6k2bm1o8b) 3

[2.4.Aplicaciones de User Space](#_5pur0xl36gqq) 3

[**3.Instrucciones de Compilación y Ejecución**](#_ezwxo2t1v5wx) **4**

[**4.Limitaciones**](#_44qec1hjr3h1) **5**

[**5.Problemas encontrados durante el desarrollo**](#_fpr2cmdek94e) **5**

[**6.Fuentes**](#_lycq6ty27fe7) **5**

# 1. Introducción

El presente trabajo práctico busca expandir sobre la base creada en el Trabajo Práctico 2, para lograr una mejor distribución de recursos. Para esto se desarrollaron mejoras a los pipes, al Physical Memory Management y al scheduler.

# 2.Implementación

## 2.1.Pipes

Los pipes fueron radicalmente modificados para poder trabajar con file descriptors, de manera que su implementación se asemeja a la de las librerías estándar de C. Partiendo del sistema de unnamed pipes que implementamos para el trabajo anterior, se expandió sobre este agregándole a los pibes y a los procesos file descriptors (para éstos se utilizó STDIN y STDOUT). Éstos funcionan como índices (0 y 1 sobre un array de enteros) para poder acceder a las operaciones de escritura y lectura sobre el buffer perteneciente al pipe. Para poder acceder a este espacio de memoria, se utiliza un sistema de 3 mutex, que le brindan al proceso que quiere acceder al buffer derecho de uso, escritura y lectura. Luego, se fija si el buffer tiene contenido, si quiere leer y está vacío o si quiere escribir y esta lleno el proceso correspondiente a la acción se bloquea hasta que se le vuelva a activar.

Además, se añadió un programa que lee constantemente del buffer del teclado y guarda e imprime en pantalla todo lo que esté contenido en él,...

## 2.2.Manejo de Memoria

Para el manejo de memoria se mantuvo el algoritmo que ya había sido implementado para el segundo trabajo práctico, el buddy allocator, que va dividiendo reiteradamente bloques de memoria a la mitad para crear dos buddies más pequeños hasta que se obtenga un bloque del tamaño deseado. Por este motivo, para que no se tenga que volver a dicho trabajo, le adjuntamos la descripción del trabajo anterior:

Cuando alocamos, verificamos si tenemos algún bloque libre del tamaño deseado. Si no, dividimos un bloque más grande tantas veces como sea necesario para obtener un bloque del tamaño adecuado.Entonces, si queremos 32 K, dividimos el bloque de 128 K en 64 K y luego dividimos uno en 32 K. Como se puede ver, este método de división significa que los tamaños de bloque siempre serán potencias de dos. Si se intenta asignar algo más pequeño, por ejemplo 13 K, la asignación se redondeará a la potencia más cercana de dos (16 K) y luego se le asignará un bloque de 16 K.

Se puede ver que hay una gran cantidad de fragmentación ocurriendo. Es importante notar que este tipo de fragmentación se llama fragmentación interna, ya que se desperdicia memoria dentro de un bloque pero no desperdicia espacio entre los bloques. Cada vez que se libera un bloque, verificamos si el buddy también es libre. De ser así, se fusionan los dos buddies en el bloque único que alguna vez fueron antes de ser separados. Esta acción se realiza recursivamente hasta que el bloque no tenga un buddy libre. La gran ventaja de este mecanismo es que se evita la fragmentación externa.

En este mecanismo de manejo de memoria, se utilizó un array con listas en donde cada lista contiene los bloques de un nivel (cada nivel tiene bloques de memoria del mismo tamaño). Hay 22 niveles en donde el nivel más grande tiene 512MB y el más chico tiene 128 bytes.

## 2.3.Scheduler

El scheduler funciona en base al armado para el trabajo anterior, solo que ahora posee distintos niveles de prioridad que pueden tener un proceso, siendo el mínimo 1 y el máximo 100 (por razones de seguridad, se sentó un máximo para evitar que el usuario ponga un valor arbitrario que haga que un proceso acapare todo el tiempo para un solo proceso). El nivel de prioridad de un proceso es encargado de representar cuántos timer ticks tiene en tiempo de ejecución antes de generarse un context switch. El scheduler también posee un contador interno que va sumando cuántos quantums pasaron desde el último context switch, y cuando el contador la prioridad del proceso corriendo, se resetea el contador y se carga el siguiente proceso en la queue (como es un algoritmo Round Robin, agarra el siguiente proceso, no filtra por prioridad). Cuando se produce el cambio de contexto, se le disminuye la prioridad al proceso para que no se termine ejecutando continuamente y produzca starvation a los demás, hasta llegr a un mínimo de 1, la excepción es que el usuario haya corrido el comando nice desde la shell para un proceso, ese proceso no disminuirá su prioridad al terminar el timer tick, dado que es un valor arbitrario que el usuario debe querer permanentemente para su proceso. El funcionamiento del comando nice será explicado en la sección de aplicaciones de usuario implementado.

Las demás funcionalidades del scheduler, como inserción y borrado de procesos a la queue, se mantuvieron similares a las del trabajo anterior.

## 2.4.Aplicaciones de User Space

* **ctrl+c**: similar al funcionamiento en BASH, presionar este conjunto de teclas finaliza la ejecución del programa que esté corriendo, matándolo instantáneamente y reiniciando la shell. Esto sucede cuando se detecta que se presionaron dichas teclas, el kernel es encargado de matar al programa que está corriendo, por lo que se vuelve a la shell.
* **philosophers**: Una solución al problema de los filósofos comensales, decidimos implementarlo de la siguiente manera, tenga en cuenta que todo el backend explicado a continuación tiene su representación gráfica en pantalla en el momento que sucede. Se decidieron que iban a haber tres estados, pensando, hambriento y comiendo, y que mediante un mutex y 7 semáforos(uno para cada filósofo, y uno de finalización), se sorteará el orden en el cual se come. Primero se procede a crear la cantidad de filósofos que se quiere, y se los crea. Éstos esperarán hasta que sea su turno de comer y luego pensar, ambos representados como esperar cierto tiempo, durante el cual no son tomados en cuenta para repartir los palillos. Cuando un filósofo no está haciendo estas cosas, está esperando para comer, y luego de esperar un rato, se vuelve hambriento, por lo que su prioridad sube y el scheduler es más propenso a darle el palillo. Los estados se marcan utilizando el mutex (que marca cuando alguien está comiendo y posee los palillos), y el acceso al filósofo mediante los semáforos. Además, el usuario posee la capacidad de hacer que el proceso padre agregue o remueva filósofos durante la ejecución, mientras más filósofos haya, más deberán esperar para comer. Para la representación gráfica, se tomó que un círculo de color sin nada adentro represente un filósofo pensando, cuando aparece una circunferencia blanca en el punto significa que el filósofo tiene hambre y está esperando comida, y finalmente un círculo blanco lleno significa que tiene comida y está comiendo.
* **messages**: Se decidió utilizar un sistema de prueba que reemplace al anterior, sin embargo es muy similar porque al fin y al cabo sigue siendo una prueba de pipes. En éste, se le pide al usuario que haga un input de 1, 2, o q, la última genera un cierre del programa, la primera le da al usuario la opción de generar un texto de hasta 50 caracteres (valor arbitrario), que una vez que se registre el enter del usuario, el programa escribe en un pipe creado para este propósito Con el 2, el usuario le especifica al programa cuantos bytes leer del pipe, los cuales una vez presionado enter éste lee del pipe y pone en pantalla (si había menos bytes de los especificados lee estos solamente).
* **pipesdemo**: éste programa tiene la función de demostrar cómo funciona la comunicación entre dos procesos simultáneamente utilizando pipes. Primero crea dos procesos que corren en simultáneo y luego sus dos pipesutilizando file descriptors, que comienzan a mandarse el mismo saludo una y otra vez, y luego leyendo el saludo del otro. Para esto, la idea es que uno le manda el saludo al otro y espera a que el otro le mande algo que leer, y viceversa con el otro.
* **nice**: el comando nice tiene dos modalidades, la primera es recibiendo como parámetro solamente el pid de un proceso, te devuelve la prioridad de dicho proceso. El otro recibiendo el pid y la prioridad, cambia la prioridad del proceso al parámetro dado.
* **priority**: la función de esta app es demostrar cómo funciona el algoritmo de scheduling, especialmente el comando nice y el algoritmo de aging (o en este caso, cómo nice lo inhabilita). Para comenzar, se tienen dos procesos corriendo en simultáneo, los dos con prioridad 1, los cuales pueden recibir una nueva prioridad tocando las teclas i (increase) o d (decrease). Ésto utiliza la syscall nice para cambiar la prioridad del proceso 1 (hay dos procesos para poder mostrar como el primero cambia en base al segundo, que se mantiene fijo en 1), lo que hace que pase más o menos quantums en el procesador. Por cada timer tick, se imprime en pantalla el proceso actual en el scheduler y su nivel de prioridad.

# 3.Instrucciones de Compilación y Ejecución

Desde el directorio del tp, corra los siguientes comandos en BASH:

**$>make clean**

**$>docker run -v ${PWD}:/root -ti agodio/itba-so:1.0**

**$>make all (en el directorio del TP)**

**$>./run.sh**

# 4.Debilidades

Con la app de pipesdemo hay un ligero problema estético, ya que para poder leer el output del programa se utiliza un delay entre mandar el mensaje y la lectura de el otro mensaje. Esto genera que una vez cortado el programa con q, quede ejecutándose la orden de wait, y que vuelva a la shell y luego imprima algo con printf, haciendo que quede desprolijo.

# 5.Problemas encontrados durante el desarrollo

Durante el desarrollo, una dificultad a superar fue el hecho de que tuvimos problemas para poder lograr el correcto uso de los file descriptors de los pipes, particularmente, el acceso a la entrada y salida estándar, y ésto llevo a que hubiese una demora en poder desarollar el comando “|” para la shell. Hubo una complicación cuando nos dimos cuenta que el programa no podía leer correctamente del buffer de input del usuario, ya que implementamos el programa que lee constantemente de este buffer.

# 6.Fuentes

Fundamentos De Sistemas Operativos - 7° Edición - Autor: Silberschatz, Galvin y Gagne.