Informe proyecto

Morata, Nahuel - Ringhetti, Franco

Sistemas Operativos - 2021 - UNS

Introducción

En este proyecto se trabajaron los conceptos vistos en la materia desde el punto de vista práctico, principalmente desde el enfoque de la resolución de problemas. Aplicamos conocimientos adquiridos a lo largo del cuatrimestre tales como: gestión, comunicación y sincronización de procesos e hilos; definiciones, conceptos, estructura y arquitecturas de sistemas operativos; administración de memoria; almacenamiento y gestión de archivos.

Procesos, threads y comunicación

Este conjunto de problemas tiene en común el énfasis en la gestión de procesos y threads; y el manejo de los posibles mecanismos de comunicación aplicables dados los requisitos del problema.

Generadores de Números

En este ejercicio debíamos trabajar un sistema compuesto por dos generadores, dos escritores, un controlador y un sincronizador. La idea general es la siguiente: los generadores (1 y 2) generan números aleatorios, el sincronizador recibe esos números y se los transmite a los escritores (1 y 2) que escriben en su correspondiente archivo; es el usuario quien determina en qué archivo se escribe mediante un ingreso en consola, que es captado por el controlador.

La interacción del usuario con el programa se hace con el teclado de la siguiente forma:

- Si ingresa un 1 se escribe en "Salida1.txt".
- Si ingresa un 2 se escribe en "Salida2.txt".
- Si ingresa un 0 termina la ejecución.
- Cualquier otro ingreso es inválido.

En términos de modelado del problema tenemos 6 procesos generados mediante llamadas al sistema fork(), el árbol de jerarquía de procesos es el siguiente:

➢ controlador,
  ↳ sincronizador
    ↳ generador 1
    ↳ generador 2
    ↳ escritor 1
    ↳ escritor 2

El controlador itera en base a los ingresos del usuario, mientras que el resto itera aparte. Cuando el usuario realiza un ingreso, el controlador dispara un mensaje (ya sea por pipe o por cola de mensaje) al sincronizador. Este último si bien realiza un chequeo por el mensaje del controlador, lo hace de forma no bloqueante, en el caso de que no reciba un ingreso continua independientemente.

Una consideración particular a tener en cuenta es que a modo de que la ejecución proceda de forma fluida, los generadores esperan al sincronizador para generar un número en vez de iterar por su cuenta. Esto no necesariamente debería ser así, es una cuestión puramente de preferencia, pero cabe destacar que cambiarlo es posible y puede hacerse sencillamente tan solo removiendo las recepciones bloqueantes de los generadores por parte del sincronizador (se hizo la prueba y funciona).

Si bien el modelo es exactamente el mismo, las diferencia entre los dos incisos radica en el mecanismo de comunicación entre los procesos:

- En el caso del primero debíamos realizarlo con pipes, para ello utilizamos un total de 7 (todos unidireccionales):
  - 1 del controlador al sincronizador, este tiene la particularidad de que la lectura por parte del sincronizador es no bloqueante. Es el encargado de transmitir la elección del usuario y cambiar el flujo de ejecución del sincronizador.
  - 2 del sincronizador a los generadores, siendo estos los opcionales dada nuestra consideración. Se encargan de modelar la notificación a los generadores de que deben producir un número. En la práctica actúan como bloqueadores, hasta que no reciben un mensaje retienen el flujo de ejecución de los generadores.
  - 2 de los generadores al sincronizador. Envían los números generados por los primeros al sincronizador.
  - 2 del sincronizador a los escritores. Les envía los números generados por los generadores a los escritores, en caso de que el flujo indique que se debe escribir en el archivo "Salida1.txt" se utiliza el pipe entre el sincronizador y el escritor 1; caso contrario se utiliza el canal entre el sincronizador y el escritor 2.

  En esta implementación, en el caso de que el ingreso del usuario sea 0, para finalizar correctamente la ejecución se debe enviar "basura" para desbloquear los pipes entre el sincronizador y los escritores.

- En el caso del segundo, debíamos realizarlo con colas de mensajes, que a diferencia del anterior la vía de comunicación era única, solo que variaba el tipo del mensaje. Algo que no tuvimos en cuenta a la hora de implementarlo era que no había necesidad de la relación padre-hijo entre los procesos, ya que sin importar su origen un proceso con solo vincularse a la cola de mensajes conociendo la clave asociada podría interactuar.

Mini Shell

En este ejercicio debíamos implementar una shell con funcionalidades básicas. Decidimos optar por un diseño modular, con un archivo principal que coordine el ingreso de comandos y flujo de ejecución; y un total de 7 archivos .c por cada comando a implementar (sin contar el .txt mostrado en la ayuda).

El archivo principal, denominado shell.c recibe el comando ingresado por el usuario, separa los parámetros, identifica el comando e invoca mediante un fork() y excecl() al comando ingresado con sus parámetros.

Los comandos implementados son los siguientes:

- crear_directorio <directorio>: Crea un directorio con el nombre pasado como parámetro en la carpeta actual, mediante la llamada al sistema mkdir().

- remover_directorio <directorio>: Remueve el directorio indicado con el nombre pasado como parámetro si no contiene archivos dentro, mediante la llamada al sistema rmdir().

- listar_directorio [directorio]: Lista el contenido del directorio con el nombre pasado como parámetro, obteniendo el directorio con la llamada opendir() y ciclando dentro mostrando su contenido con readdir().

- crear_archivo <archivo>: Crea un archivo con el nombre pasado como parámetro en la carpeta actual, mediante la llamada al sistema open().

- remover_archivo <arvhivo>: Elimina el archivo con el nombre pasado como parámetro en la carpeta actual, mediante la llamada al sistema remove().

- mostrar_archivo <archivo>: Muestra el contenido del archivo con el nombre pasado como parámetro, usando llamadas al sistema de gestión de archivos (fopen(), fgetc(), fclose(), etc).

- permisos_archivo <archivo> <permisos>: Modifica los permisos del archivo con el nombre pasado como parámetro, mediante la llamada al sistema chmod().
  - Los permisos pueden ser:
    - "r" lectura
    - "w" escritura
    - "x" ejecucion
    - "rw" lectura-escritura
    - "rx" lectura-ejecucion
    - "wx" escritura-ejecucion,
    - "rwx" lectura-escritura-ejecucion.

- help: Muestra la ayuda correspondiente al minishell, invocando a “mostrar_archivo <../sh_hel.txt>”.

- exit: Termina la ejecución del minishell.

Sincronización

Estos problemas tienen en común la implementación de mecanismos de sincronización para la resolución, ya sea por semáforos que es la más explícita o por colas de mensajes (que optamos por utilizar una suerte de mapeo de semáforos).

Planta de producción

En este ejercicio debemos simular una línea de producción en la cual la única complicación reside en una alternancia en cada iteración. En vez de ser una secuencia “ABCDEF” es primero “ABCEF” y luego “ABDEF”. Para resolver esta situación utilizamos un semáforo auxiliar al cual le hace signal() B que indicaría a C y a D que inicien, y si el turno le corresponde a uno al finalizar le cede al turno (mediante su semáforo) al otro.

Las unidades de tiempo fueron modeladas con sleep(1).

Navegando por el lago

Este ejercicio consiste en modelar un barco que transporta pasajeros desde una orilla hacia la otra de un lago. Hay tres tipos de asientos en el barco: 20 en primera, 30 en business y 50 en turista. El objetivo es coordinar la adquisición de los pasajes por parte de los pasajeros y al momento en el que el barco esté al máximo de su capacidad, simular el trayecto por el lago, descarga de los pasajeros y retorno al puerto.

El modelado del problema es el siguiente:

- Por cada clase hay un tipo de pasajero, y se crean los hilos o procesos según la cantidad de lugares disponibles (la idea es que haya más pasajeros que lugares disponibles). Por clase comparten un turno que asegura que no haya dos pasajeros de la misma clase comprando a la vez. Una vez que a un pasajero le toca el turno, se fija si hay lugares disponibles: si los hay, adquiere el ticket, cede su turno y espera a que zarpe el barco; si no, avisa que su clase está llena. Este ciclo los pasajeros lo repiten infinitas veces.

- La ticketería es la que se encarga de avisar a los pasajeros que suban. Una vez que las tres clases están ocupadas, indica a todos los pasajeros que van a zarpar; cuando el barco llegó a destino le indica a los pasajeros que deben bajarse (liberando los asientos de cada clase); y una vez que vuelven al puerto reabre indicandoles a los pasajeros que pueden comprar (mediante el turno).

A la hora de implementarlo con semáforos por cada clase tenemos: un semáforo binario que indica que la clase está llena; un mutex que simula la gestión de turnos, para que no haya más de uno comprando ticket para la clase a la vez; y un semáforo de conteo que controla la cantidad de espacios libres en cada clase. Por último tenemos un semaforo que simula que el barco zarpó.

Con respecto a la implementación con colas de mensajes, es un mapeo directo de la de semáforos. Esto se debe a que simulamos el funcionamiento de semáforos con los tipos de los mensajes y la cantidad de mensajes en la cola (que es única). Lo único que consideramos que cabe destacar es el hecho de que para simular el trywait() tuvimos que implementar un msgrcv() no bloqueante (con el flag IPC_NOWAIT) y controlar si retornaba señal de error o no, para saber si recibió o no un mensaje.

El problema que presenta la solución por semáforos es que el procesador no ve a cada hilo pasajero como una unidad para planificar sino que ve todo el proceso como uno, dejando la planificación interna a merced de la librería POSIX. Por lo que pudimos observar esta planifica de modo FIFO, ya que cuando realizamos pruebas les asignamos un id a cada hilo en función a su orden de creación, y siempre se ejecutaban de menor a mayor.

Problemas conceptuales

Ejercicio 1

Datos:

- 32 bits para direcciones
- Páginas de 1KB
- Memoria principal de 128MB
- TLB de 8 entradas

Conclusion datos:

- La memoria principal puede almacenar 2^{17} frames (128 * 1024KB = 131072 KB = 2^{17}, por $log_2(131072) = 17$)
- El direccionamiento lógico es de 2^{32} - 2^{10} = 2^{22} páginas ($log_2(1024)$ por una página, entonces es 2^{10})

a) Formato de tabla de páginas:

- Tamaño tabla:

cantidad de páginas * (bits de número de frames + bit para algoritmo de reemplazo y bit de validez) =
= 2^{22} * (17 + 1 + 1) = 79691776 bits = 9728 KB

Por lo tanto ocupa 9728 páginas.

- Campos y longitud de campos:
  - Número de frame de 17 bits
  - Bit de referencia de 1 bit
  - Bit de válido-inválido de 1 bit

Formato de TLB:

- Tamaño de TLB:

  cantidad de entradas * (bits de número de página + bits de número de frame + bit válido) =
  = 8 * (22 + 17 + 1) = 320 bits = 40 B

- Campos y longitud de campos
  - Número de página de 22 bits
  - Número de frame de 17 bits
  - Bit de válido-inválido de 1 bit

b)

- Selecciona primera víctima (0320FFH) se analiza su bit de referencia cómo está en 1 se le da una oportunidad y el bit de referencia se establece en 0.

- Selecciona segunda víctima (002068H) en orden FIFO se analiza su bit de referencia cómo está en 1 se le da una oportunidad y el bit de referencia se establece en 0.

- Selecciona segunda víctima (020025H) en orden FIFO se analiza su bit de referencia cómo está en 0 se reemplaza la página por 612872FH

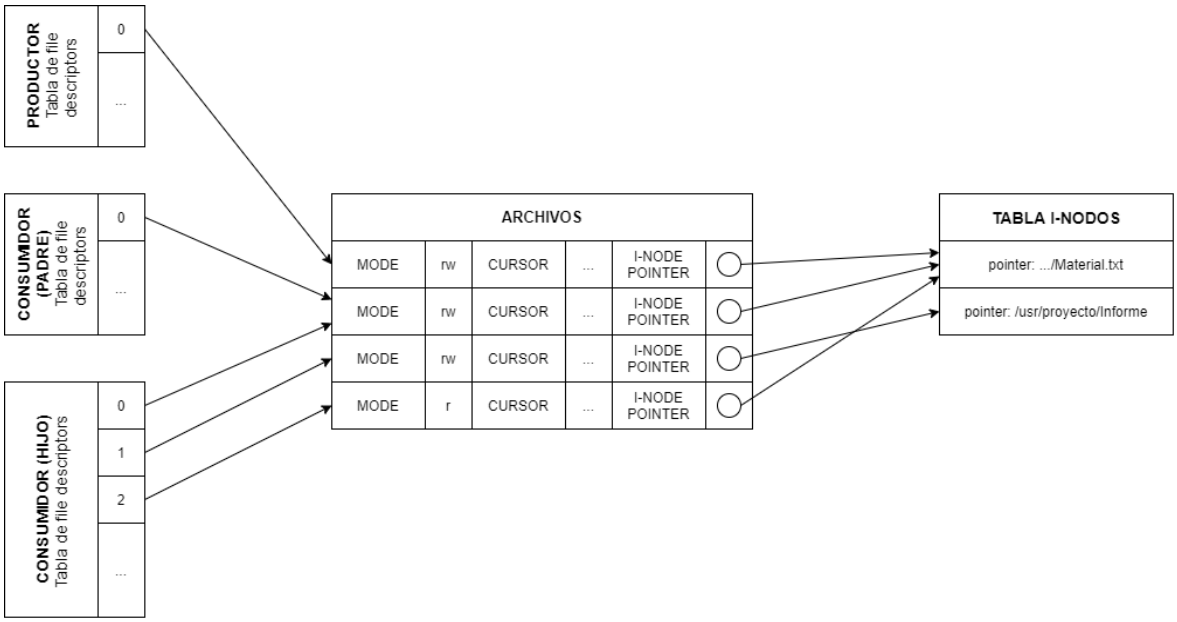
c) Cómo la dirección virtual es de 32 bits y 10 bits son de desplazamiento, entonces los 22 bits restantes se usan 8 para la tabla de primer nivel, 7 para la tabla de segundo nivel y 7 para la tabla de tercer nivel.

- 1° Nivel: 2^8 * 22 bits = 5632 bits = 704 B

- 2° Nivel: 2^7 * 22 bits = 2816 bits = 352 B

- 3° Nivel: 2^7 * (17 + 1 + 1) bits = 2432 bits = 304 B

Ejercicio 2



Conclusión

Este laboratorio nos ayudó a interiorizar los conocimientos vistos en la materia desde un enfoque práctico y nos permitió ver las aplicaciones de los conceptos que se nos presentaron desde un enfoque teórico. En el caso de sincronización, que se le dio bastante énfasis por parte de la cátedra y tanto en los prácticos como en los laboratorios se dieron problemas, los dos vistos nos parecieron más tangibles en términos de implementación en la vida real. Si bien no dejan de ser ejercicios, a comparación del problema de los fumadores o del oso y las abejas; estos nos pareció más plausible que alguien en la vida real necesite implementarlos.

Con respecto a “Navegando por el lago” originalmente, en el caso de que no hubiera más lugar, el pasajero intentaba en otra fila o no podía acceder al barco. Una corrección que tuvimos sobre esto fue que los pasajeros independientemente si hay o no lugar en el barco, pueden adquirir el ticket que deseaban solo que van a tener que esperar a que el barco retorne para poder subirse. Un problema que le detectamos a esta solución es que si no utilizamos una estructura de datos (como una cola), no hay forma de controlar que un pasajero que estaba esperando antes de que vuelva suba (que no se le colen pasajeros nuevos).