Proyecto 2021

Alumnos

Buñes Juan, 116777

Almaraz Fabricio, 109781

1.1. Procesos, threads y Comunicación

1. Generadores de números.

Comunicación con pipes

Para la solución de este problema utilizamos 5 pipes para la comunicación entre los procesos:

- **p_signal**: Se utilizó para indicar a los procesos generadores en qué momento era necesario la generación de un nuevo número.
- **p_number**: Se utilizó para comunicar al proceso sincronizador los números generados, por parte de los generadores.
- **p_write1** y **p_write2**: Ambos pipes tenían como función la comunicación entre el sincronizador y el escritor 1 y entre el sincronizador y el escritor 2 respectivamente.
- **p_ctrl**: Este pipe era el utilizado para la comunicación entre el proceso de control y el proceso de sincronización.

Se implementaron 5 métodos principales:

- **generador():** Este método es ejecutado por 2 procesos simultáneamente, ambos espera un mensaje de p_signal y en caso de recibirlo obtienen un número random, a través de un método auxiliar **random_number()**, y lo envían a través del pipe p_number.
- **sincronizador():** Este método es ejecutado por un solo proceso, el proceso padre. Este método recibe mensajes a través de 2 pipes.
  Recibe números provenientes de los generadores en el pipe p_number, y recibe (posiblemente) un mensaje de control a través del pipe p_control.
  Mantiene, además, una variable que determina a qué proceso escritor debe enviar el número recibido en p_number. En el caso de recibir un mensaje de control esta variable cambia su valor, por ende el proceso escritor de destino.
  Si no se recibe un mensaje de control, entonces se envía un mensaje a través de p_write1 o p_write2 a uno de los procesos escritores según corresponda.
- **escritor1() y escritor2()**: Método que recibe un número a través de p_write1 o p_write2, imprime la salida por consola, almacena el número en un archivo “Salida1.txt” o “Salida2.txt” y posteriormente envía un mensaje a través de p_signal a los procesos generadores para repetir el ciclo.

- **control():** Mantiene una variable con un valor entre 1 y 2, la cual cambia en un tiempo aleatorio entre 3 y 7 segundos, para posteriormente enviar el valor de la variable en un mensaje a través de p_ctrl al proceso sincronizador.

Durante la implementación de este ejercicio nos surgió un inconveniente con respecto al momento en el que el proceso control debía enviarle un mensaje al proceso sincronizador. El problema que teníamos era que como el proceso sincronizador esperaba un mensaje a través del pipe p_ctrl, entonces se quedaba esperando dicha acción y no continuaba recibiendo números de generadores y enviandolos a algún escritor.
La solución encontrada fue la utilización de las librerías **fnctl.h** y **errno.h**, a través de las cuales podemos hacer un chequeo sobre el pipe p_ctrl y en el caso de que no haya un mensaje continuar con la ejecución.

```
while (1) {
    nread = read(p_ctrl[0], &request_ctrl, SIZE_MSG);
    switch (nread) {
```

Dependiendo del valor (int) de nread el programa continúa.
Referencia: Non-blocking I/O with pipes in C

Compilado y ejecución

El archivo fuente correspondiente a la resolución de este problema es **gen_ pipes.c**, ubicado en /Generadores de números.
Dicho archivo se ejecuta por medio de un Shell Script llamado **gp.sh** ubicado en la misma carpeta. A través de una terminal localizamos el Script e ingresamos ***sh gp.sh***.

Comunicación con colas de mensajes

Este ejercicio es similar al anterior pero con la diferencia de que la comunicación es a través de colas de mensajes y que decidimos poner un número límite de generación de números para que el programa finalice correctamente y elimine previamente la cola de mensajes residual.

Tiene 4 métodos principales:

- **generador():** Primero se asocia a la cola de mensajes generada por el proceso padre. Luego obtiene un número aleatorio a través de la función auxiliar random_number(), le asigna el tipo 1 al mensaje nuevo y lo envía a la cola de mensajes.
- **sincronizador()**: Como es ejecutado por el proceso padre, entonces no se asocia a la cola de mensajes, sino que recibe el msgid por parámetro.
  Mantiene una variable destino que indica a qué proceso escritor enviar el número.
  Recibe mensajes de la cola correspondientes al tipo 1 (generadores) o al tipo 4 (control), luego dependiendo del valor de la variable destino le asigna el número a un nuevo mensaje, le asigna además un tipo y lo envía. El tipo puede ser 2 (escritor 1) o 3 (escritor 2).
  En el caso de recibir un mensaje de tipo 4 (control), entonces el valor de la variable destino cambia (de 1 a 2 o viceversa).
- **escritor()**: Este método es ejecutado por 2 procesos simultáneos correspondientes al escritor 1 y al escritor 2.
  Inicialmente se asocia a la cola de mensajes. Luego se recibe un mensaje de tipo 2 o de tipo 3 según corresponda (Al escritor 1 le corresponden los mensajes tipo 2 y al escritor 2 los tipo 3), se escribe el valor del número recibido por mensaje en un archivo (Salida1.txt o Salida2.txt) y se imprime por pantalla.
- **control()**: Tiene la misma funcionalidad que la función control() del ejercicio implementado con pipes.
  Mantiene una variable 'destino' al igual que la función sincronizador(), para almacenar el número de destino (1 o 2).
  Inicialmente se asocia a la cola de mensajes y luego en un número finito de veces cambia la variable destino entre 1 y 2.
  Le asigna el tipo 4 a un nuevo mensaje además del número destino y lo envía a la cola de mensajes.

En este ejercicio al igual que en el de pipes nos surgía el problema de cómo recibir el mensaje de control en sincronizador sin frenar la ejecución del proceso en dicho método.
En este caso la solución fue utilizar como **msgflg** en el msgrcv a IPC_NOWAIT, el cuál nos permite evaluar la existencia de mensajes del tipo que describimos en msgrcv y en el caso de que no haya nos retorna el valor -1 y nos permite seguir con la ejecución.

```
if(msgrcv(msgid, &msg_control, SIZE_MSG, 4, IPC_NOWAIT) == -1) {
```

Referencia: [msgrcv(2): message operations - Linux man page]

Compilado y ejecución

El archivo fuente correspondiente a la resolución de este problema es **gen_ messages.c**, ubicado en /Generadores de números.
Dicho archivo se ejecuta por medio de un Shell Script llamado **gm.sh** ubicado en la misma carpeta. A través de una terminal localizamos el Script e ingresamos ***sh gm.sh***.

2.Mini Shell

En este ejercicio se implementaron los siguientes comandos:

- **mkdir:** Crea un directorio
- **rmdir:** Elimina un directorio
- **create:** Crea un archivo
- **show:** Muestra el contenido de un archivo
- **dirlist:** Muestra un listado con el contenido del directorio
- **chperm:** Cambia los permisos del archivo
- **quit:** Termina la ejecución de la minishell
- **help:** Muestra una ayuda con los comandos posibles.

Por una elección de diseño se decidió implementar cada comando en un archivo distinto, para mejorar el entendimiento del código y así facilitar el trabajo.
El programa está dividido en 10 archivos, de los cuales 8 corresponden a los comandos, 1 a un header file y el último a la minishell.

El header file (shared.h) contiene todos #include y los #define, además de dos funciones que son utilizadas por más de un archivo, las cuales son:

- print_path(): Imprime la ruta actual.
- report_and_exit(): Salida en caso de errores.

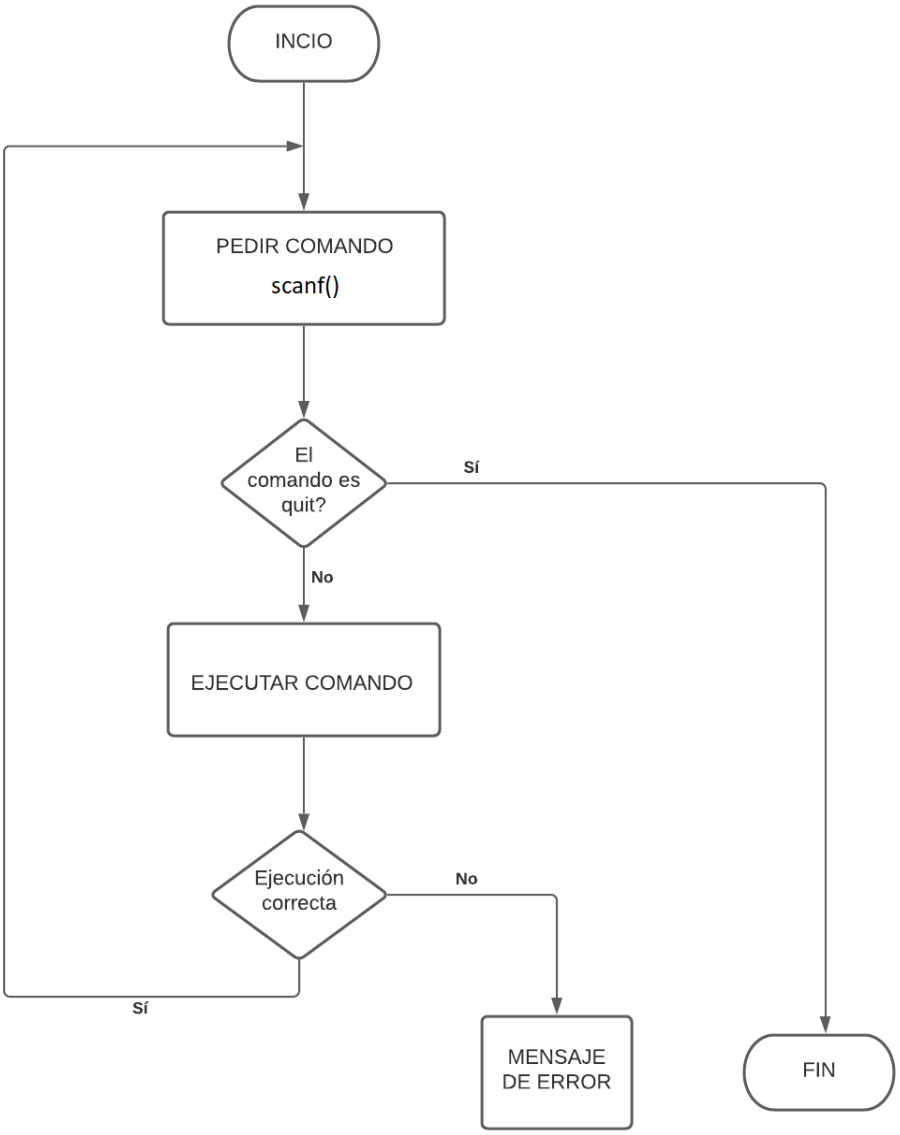
Los archivos correspondientes a los comandos: **chperm.c**, **create.c**, **dirlist.c, help.c**, **mkdir.c**, **quit.c**, **rmdir.c**, **show.c**; tienen una implementación similar. Todos están compuestos por dos funciones, las cuales son:

- main(): Inicia la ejecución del comando y recibe por parámetro los posibles argumentos.
- metodo_auxiliar(): Es llamado por el main y realiza la ejecución concreta del comando correspondiente. El nombre de este método varía en cada archivo.

El archivo principal: **minishell.c** está compuesto por tres métodos:

- get_arguments(): Retorna la cantidad de argumentos leídos en el comando pasado por parámetro y almacena dichos argumentos en un arreglo de strings.
- valid_arguments(): Analiza los argumentos ingresados y retorna el identificador del comando correspondiente. Por ejemplo, para el argumento “rmdir” si la cantidad de argumentos es > 1 entonces retorna 0 (correspondiente a RMDIR).
- main(): Método inicial de la minishell. Ejecuta un while que finaliza en el momento en el que el usuario ingresa el comando ‘quit’.
  En cada iteración recibe la entrada del usuario (comando y argumentos), y si son válidos genera un nuevo proceso hijo para la ejecución del comando correspondiente. En el caso de que no sea un comando o argumento válido, entonces muestra por pantalla un mensaje de error.

Funcionamiento de Main() en minishell.c



Compilado y ejecución

Los archivos fuente correspondientes a la resolución de este problema son **chperm.c**, **create.c**, **dirlist.c**, **help.c**, **minishell.c**, **mkdir.c**, **quit.c**, **rmdir.c**, **show.c** y **shared.h**, ubicados en /Mini-Shell.

Dicho archivo se ejecuta por medio de un Shell Script llamado **minishell.sh** ubicado en la misma carpeta. A través de una terminal localizamos el Script e ingresamos ***sh minishell.sh***.

1.2 Sincronización

1. Planta de producción.

Para la resolución de este problema se utilizaron 6 hilos y 6 semáforos para su sincronización.
El programa consta de 6 métodos principales, uno para cada elemento a ser fabricado. Cada método es ejecutado por un hilo distinto.
Inicialmente se inicializan los 6 semáforos con el valor 0, lo que no permite que los métodos de fabricación de elementos comiencen a "producir".
Los métodos fabricadores de elementos funcionan todos de igual manera, al inicio esperan por la obtención de un semáforo, luego cada uno tiene un sleep(n), para pausar la ejecución por unos segundos, con un n que depende del elemento a fabricar (entre 1 y 7); luego imprime por pantalla el elemento y habilita el semáforo del siguiente elemento a producir.
En el método productor de 'B' se realiza dos veces esta secuencia ya que al finalizar su producción debe habilitar a dos semáforos, pero de forma intercalada.

2. Navegando por el lago.

Compilado y ejecución

El archivo fuente correspondiente a la resolución de este problema es **Sincronizacion2.c**,
Dicho archivo se ejecuta por medio de un Shell Script llamado **Sincronizacion2.sh** ubicado en la misma carpeta. Es posible que se necesite ejecutar el comando: chmod +x Sincronizacion2.sh para darle permisos de ejecucion. Luego ejecutar con ./Sincronizacion2.sh

I. Para la resolución de este problema utilizando hilos y semáforos contamos con las siguientes variables:
   - **pthread_t pasajeros [100]:** Un arreglo de tamaño 100 de pthreads donde cada uno de esas componentes representa un pasajero
   - **sem_t entrada:** Sincroniza la entrada al barco estableciendo una sección crítica que permite que solo un hilo (o pasajero) entre a la vez al barco.
   - **sem_t salida:** Sincroniza la salida del barco estableciendo una sección crítica que permite que solo un hilo (o pasajero) salga a la vez del barco.
   - **sem_t sem_turista:** Determina la cantidad de espacios libres que hay para turistas en el barco
   - **sem_t sem_business:** Determina la cantidad de espacios libres que hay para clase business en el barco
   - **sem_t sem_primera:** Determina la cantidad de espacios libres que hay para clase primera en el barco

- **sem_t ventaTicket:** Permite sincronizar la venta de tickets de manera tal de que no se venda un ticket hasta que el pasajero anterior no termino de entrar al barco que es cuando se finaliza la compra

II. El funcionamiento del programa es el siguiente: primero se inicializan todos los semáforos y el arreglo de los hilos que representan a los pasajeros. Luego hay un ciclo while que cicla hasta que el barco está lleno, dentro de este ciclo se venden los tickets y los pasajeros entran al barco, para ello primero se elige un numero aleatorio entre 0 y 2 con la función rand() y basado en el tiempo del sistema, si es 0 se recibe un pasajero que quiere comprar un ticket turista, si es 1 clase business y si es 2 clase primera. Supongamos que llega un pasajero con deseos de comprar un ticket turista, lo primero que sucede es que entra en la seccion de codigo de tipo de pasajero 0, que es turista, y luego se fija si hay lugar en el barco, mediante la siguiente llamada: if(sem_trywait(&sem_turista)==0), si entra en el if, quiere decir que hay lugar y se hace un signal al semáforo **entrada** para que permita la entrada al barco a ese pasajero, luego se crea el hilo y se selecciona como rutina del hilo, la función turista. Mientras que el programa principal espera a que termine la venta del ticket, el hilo procede a ejecutar un wait al semáforo **entrada**, se realiza un sleep de 1 segundo para que la ejecución no resulte muy abarrotada y luego aparece un mensaje por consola indicando que el pasajero ingresó al barco, realiza un signal al semáforo **ventaTicket** indicando al programa principal que puede seguir con la venta de tickets y luego este hilo se queda esperando al semáforo **salida** que le indica que el barco ya llegó a destino y el pasajero puede salir del barco. Si el barco no tuviese más lugar para una determinada clase de pasajero, el pasajero se va y se hace signal del semáforo **ventaTicket** indicando que se finalizó esa venta aunque no hubo ninguna. El ciclo del programa principal ejecutará hasta que el barco está lleno y una vez que esto ocurra, se imprimirán unos mensajes por consola junto con algunos sleep() para simular el viaje del barco y luego se ejecutará un signal del semaforo **salida** para indicarles a los pasajeros que pueden empezar a salir del barco. El programa principal realiza un ciclo de 0 a 99 esperando que todos los hilos del arreglo pasajeros finalicen, y mientras tanto uno de los hilos seguirá su ejecución ya que puede pasar el sem_wait(&**salida**), mostrará un mensaje por consola, y luego realizará un signal al semáforo correspondiente a su clase indicando que dejó disponible lugar de ese tipo en el barco, y tambien realizara un signal al semaforo **salida** para que el siguiente pasajero salga del barco. Una vez todos los pasajeros salgan del barco se mostrará un mensaje, y debido a que todo lo anterior está dentro de un ciclo while(1), se repetirá el viaje del barco indefinidamente y de manera adecuada ya que los semáforos quedan con el valor que corresponde para que no haya problemas en sucesivos viajes del barco.

Algunos inconvenientes que surgieron fueron que en principio queríamos representar la cantidad de pasajeros totales que había en un barco con un semáforo,

pero como necesitábamos ese valor para determinar qué componente del arreglo pasajeros era el que seguía a ejecutarse, tuvimos que utilizar una variable local entera que no es utilizada para otra cosa que para determinar cuando el barco está lleno y quien es el próximo hilo a crearse.
Otro inconveniente surgia que al terminar el primer viaje del barco, el siguiente tenia problemas de sincronización, como por ejemplo que los pasajeros entraban y salían del barco inmediatamente sin que el barco iniciara el recorrido, esto se debía a como quedaba el valor del semáforo **salida** al finalizar el primer viaje, pero lo resolvimos al escribir un sem_wait(&**salida**) cuando el barco terminaba su viaje y estaba vacío, listo para su siguiente viaje.

III. Compilado y ejecución

El archivo fuente correspondiente a la resolución de este problema es **Sincronizacion2procesos.c**, **turista.c**, **business.c** y **primera.c**
Dicho archivo se ejecuta por medio de un Shell Script llamado **Sincronizacion2Procesos.sh** ubicado en la misma carpeta. Es posible que se necesite ejecutar el comando: chmod +x Sincronizacion2Procesos.sh para darle permisos de ejecución. Luego ejecutar con ./Sincronizacion2Procesos.sh

2.2.1 Lectura

1. Elegimos el artículo ”Allied Telesis AlliedWare Plus Operating System”.
   a. La arquitectura en la cual corre el sistema operativo AlliedWare Plus, tiene un kernel que está fundamentalmente basado en el kernel de Linux, el cual se encarga de llevar a cabo tareas como iniciar, finalizar, planificar y permitir la comunicación de los procesos, además de administrar los recursos del sistema. Si bien este sistema operativo está basado en Linux, tiene algunas modificaciones que permiten que funcione mejor en el entorno en el cual se utilizará este producto.
   Una de las prestaciones más importantes de esta arquitectura es que es modular, esto quiere decir que cada característica opera en su propio proceso en el sistema, controlado y protegido por el kernel. Este tipo de arquitectura no permite que un error de software pueda resultar en una falla total de sistema, ya que cada proceso está completamente aislado de los demás. Otro beneficio de la arquitectura modular es que cada proceso individual puede ser reiniciado sin tener que interrumpir a todo el sistema.
   Otro aspecto importante de la arquitectura de Allied Telesis, es la abstracción de hardware, esto significa que esta arquitectura puede ser soportada por distintos tipos de hardware sin tener que modificar código para adaptarse o utilizar las abstracciones que se utilizan. Esta operación es llevada a cabo por el HAL (Hardware

Abstraction Layer) quien se encarga de convertir de conceptos abstractos a términos específicos del hardware y viceversa.

Otro punto clave de la arquitectura es como se maneja el flujo de control y datos. Utilizando el HAL, es posible separar por completo dos procesos que están comunicándose. Con esto es posible que el flujo de control sea manejado por un componente del sistema mientras que los datos son manejados por otro componente del sistema completamente distinto.

b. Un elemento que es representativo para este tipo de sistemas operativos es la portabilidad el cual en este caso, se alcanza a través de ciertos componentes de la arquitectura como el HAL, a la inherente portabilidad del kernel de Linux y a la disciplina de código que se utilizó al desarrollar estos sistemas. Esta portabilidad es importante debido a que este tipo de sistemas operativos va a ser utilizado en distintos sistemas como routers o switches.

   Otro aspecto importante en estos sistemas operativos es la confiabilidad y resiliencia, ya que en sistemas que trabajan en entornos de redes se busca minimizar todo lo posible los errores para no perder información y luego tener que replicarla. En el caso del sistema operativo AlliedWare Plus, esto se alcanza en mayor parte por estar basado en un sistema confiable y resiliente como lo es Linux. Como muchas veces los errores son inevitables, un aspecto importante es cómo se actúa frente a estos errores, si es que se los deja pasar o si se toma alguna medida. Por ejemplo en el sistema de AlliedWare Plus se usan mecanismos que son útiles para un posterior debug.

   Un aspecto que quizá sea el más importante, es el nivel de seguridad que manejan este tipo de sistemas operativos. Es frecuente que estos sistemas operativos manejen información muy sensible por lo que los mecanismos de seguridad son de las cosas más importantes a tener en cuenta. En el sistema operativo que analizamos, se hace mucho énfasis en la autentificación de los usuarios. Cada uno de los usuarios debe estar identificado y existen distintos tipos de usuarios con distintos permisos para no comprometer la información del sistema. Este concepto de autentificación no solo se aplica a los usuarios sino que también se aplica a los distintos dispositivos con los que se relaciona el sistema.

c. En este artículo se presentan las distintas características que presenta el sistema operativo AlliedWare Plus, acompañado de los distintos componentes que conforman a la arquitectura en la que se desempeña. Este sistema operativo es utilizado en switches de alta gama y routers de la marca Allied Telesis. En el artículo se habla de algunos de los atributos de calidad con los que trabaja este sistema así como detalles directos de su comportamiento en el hardware que resaltan las cualidades positivas de este sistema. Por ejemplo, anteriormente mencionamos la importancia de la portabilidad en este tipo de sistemas operativos. El acercamiento del producto de Allied Telesis es el de tomar ventaja de los distintos sistemas que son

adecuados para las distintas aplicaciones de redes que existen, es por eso que tomaron la acertada decisión de basarse en un sistema como Linux para favorecer la portabilidad, pero también realizaron cambios para optimizar las estructuras de datos y cambios respecto a la replicación de datos entre distintos módulos para poder llevar su sistema operativo incluso en el hardware de gama más baja. Además, el uso de Linux, permite explotar otro punto importante que es el alcanzar el máximo rendimiento posible de cada dispositivo, incluso en los de gama más alta. Esto es debido a la capacidad del kernel de Linux de alocar distintos threads a distintos cores que se alinea con la naturaleza del diseño multi-thread del software de AlliedWare Plus. Trabajar con linux también trae otras ventajas como que es de código abierto, y que resulta familiar a la mayoría de los usuarios debido a su popularidad.

Otro atributo de calidad que posee este sistema y que se menciona en el artículo, es su escalabilidad. La motivación para perseguir este atributo es que en el ámbito de redes, resulta importante poder llevar datos de un lugar a otro cuando el hardware y software en cada uno de los puntos puede diferir en gran medida.

Otra punto que nos pareció importante resaltar, viene de la mano con el desarrollo del sistema de este artículo. Es interesante ver como un equipo de desarrolladores expertos en distintas áreas que están repartidos geográficamente por el mundo y donde el sistema se desarrolla bajo un esquema de metodologías ágiles, tanto para el desarrollo como para el testing, puede ser llevado a cabo con éxito.

Algo que nos resultó interesante de la lectura del artículo, es lo extensible que resulta el sistema operativo Linux y como puede ser modificado para acomodarse tanto a las necesidades del hardware como las del software, independientemente del ámbito en el que se trabaje. Nunca se nos hubiese ocurrido pensar que el sistema operativo de dispositivos orientados a redes, estaría basado en Linux por tantas razones que convergen en una razón principal que es que básicamente resulta muy sencillo trabajar e implementar nuevas características con ese sistema operativo.

En conclusión, en este artículo pudimos aprender acerca de los distintos factores y características que se necesitan tener en cuenta cuando se trabaja en el sector de redes, así también como un pantallazo general acerca de algunas de las tecnologías de software y hardware que utiliza concretamente el sistema de Allied Telesis.