Sistemas Operativos y Redes (E0224)
Año 2021

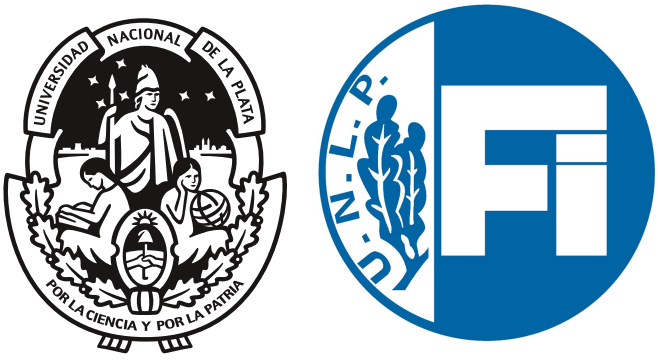
Trabajo Práctico N°4

Grupo N°4:

Ignacio Hamann - 68410/3
Juan Pablo Elisei - 68380/5
Tomás Tavella - 68371/4

Resumen

Para este trabajo se desarrollan redes.



Facultad de Ingeniería
Universidad Nacional de La Plata

Índice

1. Enunciado

En la planta industrial de una empresa, hay cuatro dependencias: Gerencia, Producción, Administración y Expedición.

Se contrata un servicio para proveer internet, y se quiere diseñar la interconexión de las dependencias asignando a cada una de ellas subredes separadas, asignando 32 direcciones IP a cada una de las subredes pertenecientes a Producción y Administración, 16 direcciones IP a cada subred de Gerencia y Expedición y además se quiere proveer WiFi en el Comedor, mediante otra subred con 64 direcciones IP.

La planta industrial tiene dos edificios separados, en el primero de ellos se ubican Producción y Expedición y en el segundo se encuentran Administración, Gerencia y el Comedor.

El proveedor de servicio de Internet, instala la conexión en el primer edificio y provee un router que desde el lado externo está conectado a la subred `198.235.150.128/25` con dirección IP texttt198.235.150.136 con *Default Gateway* tiene `198.235.150.129`. Del lado interno de la empresa, provee la subred clase C `198.235.151.0/24`, y la dirección asignada al router es `198.235.151.1`. La máscara de esta subred puede modificarse, pero no el IP del router.

1. Proponga un esquema de conexión de las distintas subredes, que emplee un router en cada uno de los edificios, y que utilice una subred diferente para interconectar ambos routers. Esta última subred debe emplear la mínima cantidad posible de direcciones IP.
2. Asigne números de subred y máscaras a cada subred. Enumere las direcciones de red y de broadcast de cada una de ellas, trate de que queden la mayor cantidad de direcciones libres para eventuales ampliaciones.
3. Asigne un *Default Gateway* a cada subred.
4. Para evitar instalar protocolos de ruteo internos, la empresa decide instalar rutas estáticas en los routers. Escriba cuales serían las tablas de ruteo necesarias en cada router, para que todos los hosts puedan alcanzar Internet, y además se puedan comunicar entre sí. Si en su diseño de red, los routers poseen más de una interfaz, enumérelas como `IF0`, `IF1`, ... , `IFN` si necesita explicitar la interfaz de salida.
5. Simule su diseño en *CORE*:
   a) En la simulación, debe mostrarse por lo menos dos hosts conectados en cada subred, excepto en el enlace entre routers.
   b) En la simulación debe ser posible observar el funcionamiento del protocolo ARP para obtener las direcciones físicas.
   c) También debe ser posible mostrar la conectividad entre los diferentes hosts de la red y con la salida a Internet mediante el uso del comando `ping`.

2. Interpretación del problema

Para este programa, se debe crear dos espacios de memoria compartida utilizando las funciones de la biblioteca `<sys/shm.h>`, en los que existirán las dos mitades de un *buffer ping-pong*.

Los datos leídos del archivo `datos.dat` se almacenan en una estructura con los siguientes elementos:

- Una variable de tipo `int` que almacena un identificador menor a 50000.
- Una etiqueta con el tiempo en el que fue escrito el dato, con precisión de micro segundos.
- El dato que se leyó del archivo, de tipo `float`.

Como no se aclara el tamaño del *buffer* en el enunciado, se elige que este tenga un tamaño total de 100 estructuras de las mencionadas previamente, dividido en dos mitades de 50 estructuras.

El proceso productor debe leer datos del archivo `datos.dat` y almacenar las estructuras en su mitad del *buffer*, mientras que el proceso consumidor debe leerlas en la otra mitad del *buffer* a medida que estén disponibles, para imprimirlas en pantalla y escribirlas en un nuevo archivo `datos.csv`.

Para administrar el *buffer* de manera que no surjan condiciones de carrera y tanto el productor como el consumidor puedan trabajar en secciones críticas sin interrupciones, se van a crear dos programas que utilizan métodos distintos para este fin:

- **Semáforos:** se utilizan los semáforos para sincronizar con las llamadas a sistema contenidas en la biblioteca `<sys/sem.h>` y memoria compartida mediante `<sys/ipc.h>`.
- **Colas de mensajes:** se utilizan las colas de mensajes para sincronizar mediante las llamadas a sistema de la biblioteca `<sys/msg.h>`.

3. Resolución

Se utilizaron las siguientes bibliotecas de C para poder llevar a cabo la resolución del problema planteado:

- Para ambas implementaciones:
  - `<sys/ipc.h>`: biblioteca de *System V* para la comunicación entre procesos.
  - `<sys/time.h>`: biblioteca para obtener el tiempo de la *timestamp*.
  - `<sys/shm.h>`: biblioteca de *System V* para la memoria compartida.
- Para la resolución con semáforos:
  - `<sys/sem.h>`: implementación de *System V* para semáforos.
- Para la resolución por colas de mensajes:
  - `<sys/msg.h>`: implementación de *System V* para colas de mensajes.

En ambos casos se eligió reservar dos segmentos de memoria compartida, uno para cada mitad del *buffer*, con capacidad para almacenar 100 estructuras de datos en total. Esto se logró mediante la función `shmget()` (contenida en la biblioteca `<sys/shm.h>`), a la cual se le pasan como argumentos el tamaño del segmento y una clave única (obtenida con la función `ftok()`) que debe ser la misma para el productor y el consumidor, para así poder compartir los segmentos creados.

Además, dado que los requerimientos no establecen que se debe hacer en el caso que el productor quiera escribir un *buffer* que aun no fue consumido, se decidió priorizar la integridad de los datos a detrimento de la velocidad de respuesta, por lo que el productor espera a que el consumidor finalize de leer los datos del textitbuffer.

3.1. Realización con semáforos

Para la implementación con semáforos, se utilizaron las funciones `semget()`, `semctl()`, `semop()` incluidas en la biblioteca apropiada para implementar 3 semáforos distintos:

- **Semáforo del *buffer* 1**: Este semáforo es bloqueado por alguno de los dos procesos al comenzar operaciones sobre el primer *buffer*, y desbloqueado al terminarlas.
- **Semáforo del *buffer* 2**: De manera similar al semaforo anterior, es bloqueado por alguno de los dos procesos al comenzar operaciones sobre el segundo buffer, y desbloqueado al terminarlas.

- **Semáforo de sincronización**: Se encarga de sincronizar los dos programas, de manera que el productor no sobrescriba datos aún no consumidos.

Primeramente, se crean los semáforos y espacios de memoria compartida (con `shmget()` y `semget()`), utilizando claves únicas obtenidas con la función `ftok()`. En este caso, tanto la memoria compartida como los semáforos se crean en el programa productor, por lo que si se llama al consumidor previo a este, se va a arrojar en pantalla un error indicando que no se pudieron obtener los recursos.

Una vez creados los semáforos, se inicializan dentro del productor mediante `semctl()`, y una vez que comienza la lectura y escritura de datos, se opera sobre los mismos mediante la función `semop` (para incrementar la legibilidad del código, las tres instrucciones requeridas para operar un semáforo se sintetizaron en un macro definido en el *header* del programa).

Finalmente, al terminar de correr el productor, se llaman a `shmctl()` y `semctl()`, para indicar al sistema que destruya los semáforos y las memorias compartidas que se crearon una vez que el ultimo programa haya dejado de utilizarlos (en este caso el consumidor).

3.1.1. Pseudocódigo

```
INICIO
    Declarar y asignar variables, macros y estructuras;
    Obtener la clave de las dos memorias compartidas y el semaforo (en el caso de no obtenerlas imprimir error);
    Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no obtenerlas imprimir error);
    Asociar el espacio de memoria compartida con un puntero (si no puede asociar imprimir error);
    Crear semaforos (si no puede crear imprimir error);
    Inicializar semaforos;
    Verificar la existencia del archivo datos.dat;
    Obtener el tiempo de UNIX inicial;
    Inicializar variables auxiliares para los buffers;
    Mientras(1){
        Bloquear el semaforo del buffer1;
        Leer una linea del archivo .dat;
        Mientras(No se llegue al fin del buffer1){
            Copiar los datos en el buffer1;
            Leer una linea del archivo .dat;
            Si(Se llega al final del archivo){
                Salir del bucle y avisar que el ultimo elemento se escribio en el buffer1;
            }
            Incrementar la variable para recorrer el buffer y la del id;
        }
        Resetear la variable que recorre el buffer;
        Desbloquear el semaforo del buffer1;
        Bloquear el semaforo del buffer2;
        Bloquear el semaforo de sincronizacion;
        Leer una linea del archivo.dat;
        Mientras(No se llegue al fin del buffer2){
            Copiar los datos en el buffer2;
            Leer una linea del archivo .dat;
            Si(Se llega al final del archivo){
                Se sale del bucle;
            }
            Incrementar la variable para recorrer el buffer y la del ID;
        }
        Si(Se llega al final del archivo){
            Se sale del bucle y avisar que el ultimo elemento se escribio en el buffer2;
        }
        Resetear la variable que recorre el buffer;
        Desbloquear el semaforo del buffer2;
        Bloquear el semaforo de sincronizacion;
    }
    Poner un ID Null despues de llegar al ultimo elemento del buffer correspondiente para avisarle al consumidor que se llego al EOF;
    Cerrar el archivo .dat;
    Liberar la memoria compartida y se remover los semaforos;
FIN

Consumidor:
INICIO
    Declarar y asignar variables, macros y estructuras;
    Obtener la clave de las dos memorias compartidas y el semaforo (en el caso de no obtenerlas imprimir error);
    Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no obtenerlas imprimir error);
    Asociar el espacio de memoria compartida con un puntero (si no puede asociar imprimir error);
    Crear semaforos (si no puede crear imprimir error);
```

```
        Inicializar de semaforos;
        Verificar la existencia del archivo datos.dat;
        Inicializar variables auxiliares para los buffers;
        Mientras(1){
            Bloquear el semaforo del Buffer1;
            Mientras(No se llega al fin del buffer1 y el ID del mismo sea distinto a -1){
                Copiar los datos del buffer1 en el archivo csv e imprimirlos en pantalla;
                Incrementar la variable para recorrer el buffer;
                Si(Se encontro el ID -1){
                    Salir del bucle;
                }
            }
            Resetear la variable que recorre el buffer;
            Si(Se encontro el ID -1){
                Salir del bucle principal;
            }
            Desbloquear el semaforo del buffer1;
            Desbloquear el semaforo de sincronizacion;
            Bloquear el semaforo del buffer2;
            Mientras(No se llegue al fin del buffer2 y el ID del mismo sea distinto a -1){
                Copiar los datos del buffer2 en el archivo csv e imprimirlos en pantalla;
                Incrementar la variable para recorrer el buffer;
                Si(Se encontro el ID -1){
                    Salir del bucle;
                }
            }
            Resetear la variable que recorre el buffer;
            Si(Se encontro el ID -1){
                Salir del bucle principal;
            }
            Desbloquear el semaforo del buffer2;
            Desbloquear el semaforo de sincronizacion;
            Si(Se encontro el ID -1){
                Salir del bucle principal;
            }
        }
        Cerrar el archivo .csv;
        Liberar la memoria compartida y remover los semaforos;
    FIN
```

3.2. Realización con colas de mensajes

La implementación por cola de mensajes emplea dos colas con dos tipos de mensajes. Se utiliza una *queue* por buffer, y los tipos de mensaje indican si el buffer está listo para ser escrito, o para ser leído. El productor siempre envía mensajes de un tipo, y el consumidor envía mensajes del otro. Para la sincronización, ambos programas esperan primero a recibir el mensaje del tipo que corresponda antes de operar con la memoria compartida, y una vez que terminan envían el mensaje recíproco. Al iniciar, el consumidor es el primero en enviar mensajes de que los buffers están disponibles, con lo cual ningún programa empieza por separado, sino que el trabajo sobre la memoria compartida se realiza solo con los dos programas en ejecución.

3.2.1. Pseudocódigo

```
    INICIO
        Declarar y asignar variables, macros y estructuras;
        Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que no las obtenga imprime error);
        Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no las obtenga imprime error);
        Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime error);
        Crear colas de mensajes (si no los puede crear imprime error);
        Verificar de la existencia del archivo datos.dat;
        Obtener el tiempo de UNIX inicial;
        Inicializar variables auxiliares para los buffers;
        Mientras(no es el fin del archivo){
            Esperar mensaje que se puede escribir el buffer 1;
            Llenar buffer con datos;
            Preparar mensaje de "listo para leer";
            Enviar mensaje en cola 1;

            Esperar mensaje que se puede escribir el buffer 2;
            Llenar buffer con datos;
            Preparar mensaje de "listo para leer";
            Enviar mensaje en cola 2;
        }
        Cerrar archivo;
        Liberar memoria compartida;
    FIN

    Consumidor:
    INICIO
```

```
122         Declarar y asignar variables, macros y estructuras;
123         Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
        no las obtenga imprime error);
124         Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
        las obtenga imprime error);
125         Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
        error);
126         Crear colas de mensajes (si no los puede crear imprime error);
127         Enviar mensaje de "ambos buffers listos para escribir";
128         Abrir archivo de destino en modo escritura;
129         Mientras(1){
130             Esperar mensaje que se puede leer el buffer 1;
131             Si el mensaje indicaba fin del archivo, salir;
132             Leer buffer, imprimir en pantalla y en archivo de destino;
133             Preparar mensaje de "listo para escribir";
134             Enviar mensaje en cola 1;
135
136             Esperar mensaje que se puede leer el buffer 2;
137             Si el mensaje indicaba fin del archivo, salir;
138             Leer buffer, imprimir en pantalla y en archivo de destino;
139             Preparar mensaje de "listo para escribir";
140             Enviar mensaje en cola 2;
141         }
142         Cerrar archivo;
143         Liberar memoria compartida;
144         Liberar colas de mensajes;
145     FIN
```

3.3. Compilación y ejecución de los programas

Con el código completo, para compilar los programas a un archivo binario ejecutable se llama al comando `gcc` (*GNU C Compiler*):

```
$ gcc productor_sem.c -o prod_sem
$ gcc consumidor_sem.c -o cons_sem
$ gcc productor_col.c -o prod_col
$ gcc consumidor_col.c -o cons_col
```

Con lo que se obtienen cuatro archivos binarios ejecutables, los cuales, estando situados en la carpeta en la que se encuentran, se ejecutan desde la terminal de la siguiente manera:

```
$ ./prod_sem
$ ./cons_sem
$ ./prod_col
$ ./cons_col
```

Es importante ejecutar los productores antes que los consumidores, para cualquiera de las dos implementaciones, por la forma en que se maneja la sincronización.

4. Conclusiones

La implementación del *buffer ping-pong* con distintos métodos de sincronización para la memoria compartida demostró que si bien ambos métodos son efectivos para evitar la pérdida de datos o las condiciones de carrera, las distintas opciones tienen sus ventajas y desventajas. Las colas de mensaje permiten compartir mucha mas información entre procesos para generar distintos comportamientos deseados, aunque tienen la desventaja que al eliminarse el mensaje de la cola cuando se lee, comunicar más de dos procesos se vuelve más complicado. Los semáforos tienen un funcionamiento más sencillo, que permite sincronizar más procesos con múltiples juegos de recursos compartidos, aunque su implementación no resulta tan simple en la práctica.