

# Sistemas Operativos y Redes (E0224)

## Año 2021

### Trabajo Práctico Integrador

#### Grupo N°4:

Ignacio Hamann - 68410/3

Juan Pablo Elisei - 68380/5

Tomás Tavella - 68371/4

#### Resumen

En este informe se desarrollan los detalles de diseño e implementación de un juego de naipes españoles “Escoba de 15” en el lenguaje de programación C (y bibliotecas estándar de Linux). Con este fin, se utiliza un servidor concurrente a base de *sockets* de red que acepta hasta 4 conexiones (jugadores) simultáneos conectados bajo el protocolo TCP.



*Facultad de Ingeniería*  
*Universidad Nacional de La Plata*

# Índice

<b>1. Enunciado</b>	<b>2</b>
1.1. Manejo de errores . . . . .	2
1.2. Otras consideraciones . . . . .	2
<b>2. Interpretación del problema</b>	<b>3</b>
<b>3. Resolución</b>	<b>3</b>
3.1. Realización con semáforos . . . . .	3
3.1.1. Pseudocódigo . . . . .	4
3.2. Realización con colas de mensajes . . . . .	5
3.2.1. Pseudocódigo . . . . .	5
3.3. Compilación y ejecución de los programas . . . . .	6
<b>4. Conclusiones</b>	<b>6</b>

# 1. Enunciado

Se desea realizar un servidor TCP que permita jugar una partida del tradicional juego de cartas “Escoba de 15”, desde un cliente `telnet`.

El servidor debe esperar conexiones entrantes desde el port 1234 y deberá incorporar, como mínimo, las siguientes capacidades:

- a) Permitir jugar una partida entre 2, 3 o 4 jugadores.
- b) Cuando se conecte el primer jugador, se deberá ofrecer la posibilidad de seleccionar si la partida aceptará 2, 3 o 4 jugadores.
- c) Una vez seleccionada la cantidad de jugadores, deberá crear e inicializar los recursos necesarios y esperar que se presenten el resto de los jugadores.
- d) Una vez que se hayan conectado el resto de los jugadores, se repartirán las cartas y se avisará al primer jugador conectado que es el que inicia la partida (“mano”). El orden de participación del resto de los jugadores deberá ser el mismo que el orden de conexión.
- e) El servidor enviará a cada jugador conectado la información sobre cuales naipes le tocaron en el reparto y además cuales son los naipes que están sobre la mesa.
- f) El servidor habilitará al jugador que tiene el turno de juego a enviar su jugada, una vez recibida la reenviará a todos los jugadores.
- g) Luego informará a cada jugador cuáles son los naipes que tiene en su poder, cuáles son los naipes que quedan en la mesa y quién tiene el próximo turno.
- h) Si un jugador intenta enviar su jugada cuando no le toque el turno, el servidor ignorará el intento.
- i) El juego finaliza cuando no hay más cartas para repartir.
- j) Cuando finaliza el juego, el servidor informa a todos los jugadores, las cartas recolectadas por cada jugador y la cantidad de escobas para poder realizar un recuento manual de puntaje.

La descripción anterior es general, y puede implementarse de la manera que se desee, teniendo en cuenta que los clientes `telnet` o `nc` mostraran en pantalla solamente lo que reciban sin realizar ningún formateo. Se sugiere que la información a los clientes sea enviada en formato de texto.

## 1.1. Manejo de errores

Si un jugador intenta levantar un conjunto de cartas que se exceden de 15, se anula la jugada y se le indica que la comience de nuevo. Si intenta levantar dos veces la misma carta, le indica que no es válido y le pide que levante otra carta.

## 1.2. Otras consideraciones

Este juego tiene muchas variantes, pero en este caso se trata de utilizar las reglas mas sencillas. Tener en cuenta que el interés de la cátedra es que apliquen los conocimientos sobre TCP/IP y comunicaciones entre procesos.

El servidor debe ser un servidor concurrente, donde a medida que se conectan los distintos jugadores. se crea un hijo para atender a cada jugador. Los hijos deben comunicarse entre sí, mediante mecanismos de IPC como memoria compartida, colas de mensajes o semáforos.

## 2. Interpretación del problema

Para este programa, se debe crear un servidor concurrente que permita jugar una partida de la escoba de 15.

Los clientes deben conectarse mediante netcat o telnet, así que el formateo de los mensajes debe realizarse del lado del servidor. Todos los clientes deben ser atendidos por hijos, siendo el primero el que determina la cantidad de jugadores de la partida.

Debe usarse algún método de sincronización o IPC para la comunicación entre hijos y determinación de los turnos de juego, ignorando jugadas no válidas por parte de los demás jugadores.

Al finalizar la partida, el servidor quedará a la espera de nuevos jugadores para iniciar una nueva partida.

## 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 finalice de leer los datos del *buffer*.

### 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

```
1  Productor:
2  INICIO
3      Declarar y asignar variables, macros y estructuras;
4      Obtener la clave de las dos memorias compartidas y el semaforo (en el caso de no
      obtenerlas imprimir error);
5      Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
      obtenerlas imprimir error);
6      Asociar el espacio de memoria compartida con un puntero (si no puede asociar imprimir
      error);
7      Crear semaforos (si no puede crear imprimir error);
8      Inicializar semaforos;
9      Verificar la existencia del archivo datos.dat;
10     Obtener el tiempo de UNIX inicial;
11     Inicializar variables auxiliares para los buffers;
12     Mientras(1){
13         Bloquear el semaforo del buffer1;
14         Leer una linea del archivo .dat;
15         Mientras(No se llegue al fin del buffer1){
16             Copiar los datos en el buffer1;
17             Leer una linea del archivo .dat;
18             Si(Se llega al final del archivo){
19                 Salir del bucle y avisar que el ultimo elemento se escribio en el buffer1;
20             }
21             Incrementar la variable para recorrer el buffer y la del id;
22         }
23         Resetear la variable que recorre el buffer;
24         Desbloquear el semaforo del buffer1;
25         Bloquear el semaforo del buffer2;
26         Bloquear el semaforo de sincronizacion;
27         Leer una linea del archivo.dat;
28         Mientras(No se llegue al fin del buffer2){
29             Copiar los datos en el buffer2;
30             Leer una linea del archivo .dat;
31             Si(Se llega al final del archivo){
32                 Se sale del bucle;
33             }
34             Incrementar la variable para recorrer el buffer y la del ID;
35         }
36         Si(Se llega al final del archivo){
37             Se sale del bucle y avisar que el ultimo elemento se escribio en el buffer2;
38         }
39         Resetear la variable que recorre el buffer;
40         Desbloquear el semaforo del buffer2;
41         Bloquear el semaforo de sincronizacion;
42     }
43     Poner un ID Null despues de llegar al ultimo elemento del buffer correspondiente para
      avisarle al consumidor que se lleo al EOF;
44     Cerrar el archivo .dat;
45     Liberar la memoria compartida y se remover los semaforos;
46  FIN
47
48  Consumidor:
49  INICIO
50      Declarar y asignar variables, macros y estructuras;
51      Obtener la clave de las dos memorias compartidas y el semaforo (en el caso de no
      obtenerlas imprimir error);
52      Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
      obtenerlas imprimir error);
53      Asociar el espacio de memoria compartida con un puntero (si no puede asociar imprimir
      error);
54      Crear semaforos (si no puede crear imprimir error);
55      Inicializar de semaforos;
56      Verificar la existencia del archivo datos.dat;
57      Inicializar variables auxiliares para los buffers;
58      Mientras(1){
59          Bloquear el semaforo del Buffer1;
60          Mientras(No se llega al fin del buffer1 y el ID del mismo sea distinto a -1){
61              Copiar los datos del buffer1 en el archivo csv e imprimirlos en pantalla;
62              Incrementar la variable para recorrer el buffer;
63              Si(Se encontro el ID -1){
```

```

64         Salir del bucle;
65     }
66 }
67 Resetear la variable que recorre el buffer;
68 Si(Se encontro el ID -1){
69     Salir del bucle principal;
70 }
71 Desbloquear el semaforo del buffer1;
72 Desbloquear el semaforo de sincronizacion;
73 Bloquear el semaforo del buffer2;
74 Mientras(No se llegue al fin del buffer2 y el ID del mismo sea distinto a -1){
75     Copiar los datos del buffer2 en el archivo csv e imprimirlos en pantalla;
76     Incrementar la variable para recorrer el buffer;
77     Si(Se encontro el ID -1){
78         Salir del bucle;
79     }
80 }
81 Resetear la variable que recorre el buffer;
82 Si(Se encontro el ID -1){
83     Salir del bucle principal;
84 }
85 Desbloquear el semaforo del buffer2;
86 Desbloquear el semaforo de sincronizacion;
87 Si(Se encontro el ID -1){
88     Salir del bucle principal;
89 }
90 }
91 Cerrar el archivo .csv;
92 Liberar la memoria compartida y remover los semaforos;
93 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

```

1  Productor:
2  INICIO
3      Declarar y asignar variables, macros y estructuras;
4      Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
5      no las obtenga imprime error);
6      Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
7      las obtenga imprime error);
8      Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
9      error);
10     Crear colas de mensajes (si no los puede crear imprime error);
11     Verificar de la existencia del archivo datos.dat;
12     Obtener el tiempo de UNIX inicial;
13     Inicializar variables auxiliares para los buffers;
14     Mientras(no es el fin del archivo){
15         Esperar mensaje que se puede escribir el buffer 1;
16         Llenar buffer con datos;
17         Preparar mensaje de "listo para leer";
18         Enviar mensaje en cola 1;
19
20         Esperar mensaje que se puede escribir el buffer 2;
21         Llenar buffer con datos;
22         Preparar mensaje de "listo para leer";
23         Enviar mensaje en cola 2;
24     }
25     Cerrar archivo;
26     Liberar memoria compartida;
27 FIN
28
29 Consumidor:
30 INICIO
31     Declarar y asignar variables, macros y estructuras;
32     Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
33     no las obtenga imprime error);
34     Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
35     las obtenga imprime error);
36     Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
37     error);
38     Crear colas de mensajes (si no los puede crear imprime error);

```

```

33     Enviar mensaje de "ambos buffers listos para escribir";
34     Abrir archivo de destino en modo escritura;
35     Mientras(1){
36         Esperar mensaje que se puede leer el buffer 1;
37         Si el mensaje indicaba fin del archivo, salir;
38         Leer buffer, imprimir en pantalla y en archivo de destino;
39         Preparar mensaje de "listo para escribir";
40         Enviar mensaje en cola 1;
41
42         Esperar mensaje que se puede leer el buffer 2;
43         Si el mensaje indicaba fin del archivo, salir;
44         Leer buffer, imprimir en pantalla y en archivo de destino;
45         Preparar mensaje de "listo para escribir";
46         Enviar mensaje en cola 2;
47     }
48     Cerrar archivo;
49     Liberar memoria compartida;
50     Liberar colas de mensajes;
51     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 más 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.