P3. COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Sistemas Operativos 1
Abril 2023

1 EL PARADIGMA PRODUCTOR-CONSUMIDOR

Durante el desarrollo de esta práctica nos centraremos en utilizar algunos de los métodos que nos ofrece sistema operativo para comunicar procesos entre sí. Existen múltiples métodos de comunicación entre procesos entre los que podemos mencionar las tuberías, los archivos, la red así como las señales (como se ha visto en teoría).

En esta práctica se desarrollará un esquema de productor-consumidor. Este paradigma es un ejemplo clásico de sincronización de procesos. Los productores generan información y la escriben en un búfer, entregándosela a los consumidores. Los consumidores leen los datos del búfer y realizan alguna operación sobre ellos. A la hora de implementar este esquema es necesario asegurar que el consumidor no intente tomar datos si el búfer está vacío. De forma similar, el productor no puede introducir nuevos datos en el búfer hasta que el consumidor haya procesado los anteriores.

Pese a que la presente práctica se basa en el paradigma productor-consumidor, se modifica un poco el esquema de funcionamiento para facilitar la sincronización de los productores y los consumidores. En la asignatura de Sistemas Operativos 2 se volverá a analizar con mayor detalle este esquema cuando se estudie el tema sobre programación concurrente.

2 DESCRIPCIÓN DE LA PRÁCTICA

En esta práctica se propone implementar el popular juego de “Piedra, papel o tijera”[1] siguiendo el paradigma del productor-consumidor. “Piedra, papel o tijera” es un juego de manos estructurado en turnos en el que participan dos jugadores. En cada turno, cada jugador escoge uno de estos tres elementos y lo enfrenta a la elección del otro, decidiéndose quién es el ganador a través de reglas muy sencillas:

- La piedra *rompe* las tijeras (gana la piedra).
- El papel *envuelve* a la piedra (gana el papel).
- Las tijeras *cortan* el papel (ganan las tijeras).

Cualquier otra combinación distinta de las anteriores supondrá un empate.

El juego de “Piedra, papel o tijera” puede implementarse siguiendo un esquema productor-consumidor. Específicamente, se propone utilizar un único consumidor y dos productores. Cada productor representará a cada uno de los dos jugadores y el consumidor implementa el árbitro. El juego se desarrolla en una serie de turnos, en cada uno de los cuales, los productores transmiten al consumidor cuál de tres posibles opciones (piedra, papel o tijera) ha elegido. Dicha elección se llevará a cabo de forma aleatoria. El consumidor leerá las opciones y dictaminará cuál de los dos jugadores ha sido el ganador del turno o si ha habido un empate. Al final de todos los turnos, el consumidor imprimirá un mensaje por pantalla contabilizando las victorias que ha obtenido cada jugador.

Por otro lado, el programa cargará información relacionada con el juego a partir de dos ficheros. En particular, las posibles opciones que tiene cada jugador vendrán dadas en un fichero de extensión csv (comma separated values) que se adjunta en el material de la práctica

[1] https://es.wikipedia.org/wiki/Piedra,_papel_o_tijera

`opciones.csv`. Por otro lado, se proporciona otro fichero, `reglas.csv`, del mismo tipo que el anterior y que se usará para establecer las combinaciones de opciones ganadoras. Dichas combinaciones son las que se han descrito al inicio de esta sección.

El programa a implementar ha de ejecutar un total de N turnos, valor que se suministrará como parámetro de entrada. Para que el código pueda funcionar correctamente, será necesario definir un mecanismo que permita que el consumidor notifique a los productores el inicio de cada nuevo turno. En este caso el mecanismo de notificación (o sincronización) se realiza mediante señales, y más concretamente a través de la señal `SIGUSR1`. Por otro lado, también se necesitará un búfer de comunicación entre el consumidor y los dos productores, a fin de que cada productor transmita al consumidor la opción que ha escogido en cada turno. Con este propósito, se utilizará una tubería sin nombre. Se proporciona un esquema de los principales elementos de la implementación solicitada en la Figura 1.

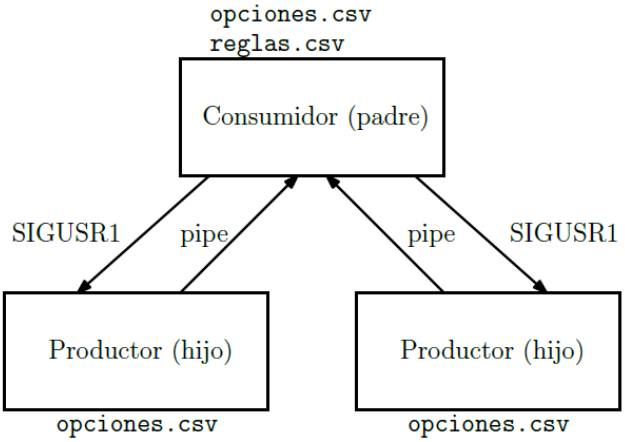

Figura 1. *Esquema del productor-consumidor a implementar. Hay un consumidor (proceso padre) Esquema del productor-consumidor a implementar. Hay un consumidor (proceso padre) y dos productores (los dos procesos hijos). La sincronización entre los procesos se realiza mediante la señal SIGUSR1. El búfer de comunicación será una tubería sin nombre. Además, el programa necesita cargar dos ficheros de datos: opciones.csv y reglas.csv. Ver el texto para más detalles.*

Cuando se hayan consumido los N turnos del juego, el consumidor imprimirá un mensaje informando de la cantidad de victorias alcanzadas por cada jugador y el programa terminará. Como curiosidad, pese a la sencillez de "Piedra, papel o tijera", es interesante su análisis desde la perspectiva de la teoría de juegos. La práctica no se centra en este aspecto, ya que únicamente aprovecharemos la estructura del juego para trabajar con conceptos de sistemas operativos y los jugadores (productores) decidirán aleatoriamente entre las distintas opciones, lo que matemáticamente constituye la forma óptima de jugar. No obstante, se han diseñado algoritmos que tratan de sacar ventaja de las tendencias que se observan cuando el juego es jugado por humanos. Estos algoritmos incluso han sido testados en competiciones de programación relacionadas con el juego[2].

3 ALGORITMO A IMPLEMENTAR

En la Figura 1 puede verse el esquema de los procesos involucrados en el paradigma del productor-consumidor. El padre es el consumidor, mientras que habrá dos hijos, los cuales harán la función de productores. La implementación debe hacerse de manera que el valor de N sea

[2] http://www.rpscontest.com/

conocido tanto por el padre como por los hijos. A continuación, se detalla el algoritmo a implementar:

1. El programa lee los parámetros de entrada, a saber, las rutas de los cheros de opciones y reglas, así como el número de turnos a ejecutar, N.
2. Se procesan los ficheros de opciones y reglas, generándose dos arrays cuyos elementos son de tipo estructurado (los detalles se pueden consultar en la plantilla de código proporcionada para la práctica, ver Sección 4).
3. Se crea una tubería sin nombre mediante la función `pipe`.
4. Mediante `signal`, se registra la función para manejar la señal `SIGUSR1`.
5. Se crean los dos productores, para lo cual se hace uso de la función `fork`. Se recomienda poner el código del productor en una función. Dicha función ha de llevar a cabo la siguiente:
    a. Inicializar la semilla del generador de números aleatorios mediante srand, que espera como parámetro un número que debería ser especíco del productor (ver Sección 4.3 para más detalles).
    b. Ejecutar un bucle de N iteraciones. En cada una de ellas es necesario:
        i. Esperar a que el proceso reciba la señal `SIGUSR1`.
        ii. Una vez recibida, la señal, el productor genera un número aleatorio con la función `rand` representando la opción escogida en cada turno.
        iii. Por último, la opción generada se ha de transmitir por la tubería mediante la función `write`.
6. Se ejecuta el código del consumidor, para el que también se aconseja crear una función específica. El código a incorporar dentro de la función debería llevar a cabo los pasos siguientes:
    a. Ejecutar un bucle de N iteraciones. A continuación se muestran las acciones que deben llevarse a cabo en cada una de ellas:
        i. Se envía la señal `SIGUSR1` de inicio de turno al primer jugador y a continuación se lee la opción que ha escogido a partir de la tubería usando la función `read`.
        ii. El proceso anterior se repite para el segundo jugador.
        iii. Se determina quién es el ganador del turno o si ha habido un empate, actualizando los contadores de victorias para cada jugador.
    b. Una vez acabadas las iteraciones, se ha de imprimir por pantalla la información sobre el total de victorias de cada uno de los jugadores.
7. Antes de acabar el programa, se cierran los dos descriptores de fichero asociados a la tubería creada con `pipe`.

4 AYUDA PARA LA IMPLEMENTACIÓN

Junto a esta práctica se entrega una plantilla a partir de la cual se puede empezar a trabajar. Esta plantilla incluye funciones para extraer la información sobre las opciones y reglas del juego dada en forma de ficheros csv. En la función `main`, se leen todas las líneas de los archivos de datos, obteniéndose un array con las distintas opciones y otro con las reglas.

Una vez hecho lo anterior, el siguiente paso sería comenzar a añadir el código al final de la función `main` de manera que se implemente el juego de “Piedra, papel o tijera” siguiendo el paradigma del productor-consumidor tal y como se ha descrito anteriormente (ver Figura 1). Se indican a continuación varios aspectos a tener en cuenta a la hora de implementar la práctica.

4.1 EJEMPLO DE SALIDA DE PROGRAMA

A continuación se muestra un ejemplo de la salida que deberá producir el programa para un turno del juego.

```
* TURNO 2
El Jugador 1 ha escogido Tijeras
El Jugador 2 ha escogido Papel
Tijeras cortan Papel
Gana el Jugador 1!
```

4.2 SINCRONIZACIÓN MEDIANTE SEÑALES

La sincronización entre procesos se realizará mediante la señal `SIGUSR1`, que el consumidor enviará a los productores mediante la función `kill`. Recuérdese que el consumidor envía la señal a los productores para indicarles que ha comenzado un nuevo turno y que deben escoger cada uno una opción y enviarla al consumidor a través de la tubería.

Al realizar pruebas con la implementación del código propuesto pueden encontrarse problemas si el mecanismo de sincronización entre procesos utiliza la función `pause`, que bloquea un proceso hasta que recibe y atiende una señal. La razón es que existe la posibilidad de que la señal sea recibida y atendida antes de que se ejecute `pause`, lo que causaría que el proceso quedase bloqueado. Una posible solución es utilizar la denominada espera activa (en inglés *busy waiting* o *spinning*) para gestionar la sincronización entre los productores y el consumidor en lugar de utilizar la función pause. La espera activa implica un código similar al que se muestra a continuación:

```
while (!sigusr1) {};
segusr1 = 0;
```

En el ejemplo anterior, el código espera consumiendo ciclos de CPU y sin bloquearse, con objeto de recibir la señal `SIGUSR1`. Supongamos que `sigusr1` es una variable global inicializada a 0, y que al recibir la señal la función que gestiona las señales pone `sigusr1` a 1. En el código propuesto no se hace ninguna llamada a `pause`: al recibir la señal se pone la variable `sigusr1` a 1 y el código sale del bucle `while` para volverla a poner a 0 de forma inmediata. Obsérvese que el código funcionará aunque se reciba (y se procese) la señal antes de entrar a la espera activa.

La solución propuesta puede ser aceptable en este caso, dado que sabemos que los consumidores no tendrán que esperar mucho para recibir la señal por parte del productor. Por tanto, no se desperdiciará demasiado tiempo de CPU[3].

4.3 TRANSFERENCIA DE DATOS AL CONSUMIDOR

En esta sección se describe cómo implementar la transferencia de datos desde el productor al consumidor en cada turno del juego según se mencionaba en el algoritmo descrito en la Sección 3. Específicamente, cada productor será tratado por el consumidor de forma secuencial. Empezando por el primer productor, el consumidor le enviará la señal SIGUSR1 que indica al productor que elija una opción y se la envíe. Después de esto, el consumidor leerá la opción y repetirá el proceso anterior para el segundo productor, concluyendo el turno.

El envío y recepción de las opciones escogidas por los productores se llevará a cabo mediante una tubería sin nombre definida mediante la función `pipe`. A `pipe` se le pasará como parámetro un array de dos posiciones, que tras la ejecución contendrá en la primera posición el descriptor

[3] No obstante, la sincronización también puede hacerse sin usar espera activa, según se propone en el ejercicio adicional de esta práctica descrito en la Sección 5.2.

que se utiliza para lectura, y en la segunda posición el que se usará para escritura. El descriptor de lectura será usado por el consumidor, quien podrá leer cada opción por medio de la función `read`. Por otro lado, el descriptor de escritura será utilizado por los productores para enviar la opción escogida mediante la función `write`.

4.4 SELECCIÓN ALEATORIA DE LAS OPCIONES

La selección de las opciones puede parecer trivial a simple vista, ya que tan sólo es necesario generar aleatoriamente un número entre tres posibles. Dado el fichero `opciones.csv`, estos números serían el 0, el 1 y el 2 (piedra, papel o tijera, respectivamente). Este objetivo se puede conseguir por medio de la función `rand`[4]. No obstante, si la semilla de generación de números aleatorios no se establece apropiadamente por medio de la función `srand`, los dos productores generarían siempre la misma opción, ya que ambos parten de copias idénticas del espacio de memoria del proceso padre.

Típicamente, a `srand` se le proporciona como parámetro de entrada el tiempo del sistema, que se obtiene mediante la llamada `time(NULL)`. En particular, la llamada a `time` devuelve la cantidad de segundos transcurrida desde una fecha determinada considerada como referencia hasta el momento actual. Sin embargo, en nuestro caso esta estrategia no sería válida, ya que ambos productores se crean prácticamente a la vez, lo que desemboca en una semilla idéntica para ambos productores.

Cualquier solución al problema que se acaba de exponer, pasa por inicializar la semilla de números aleatorios con un número que dependa del productor. Una posible vía para conseguirlo sería basarse en la función `getpid`, que devuelve un identificador numérico del proceso que la ejecuta.

4.5 DEPURACIÓN DEL CÓDIGO

Es habitual utilizar las llamadas a `printf` para depurar el código. Aunque puede ser útil en algunos casos, es conveniente mencionar que la introducción de estas instrucciones puede hacer que cambien los puntos en que los procesos productores y consumidor hagan un cambio de contexto. Por tanto, partiendo de una sincronización erróneamente implementada, es posible que añadiendo instrucciones `printf` el código funcione correctamente mientras que eliminándolas no lo haga. Será necesario por tanto analizar con detalle el código para encontrar la fuente del problema.

5 EJERCICIO OPCIONALES

Se proponen dos ejercicios opcionales para complementar el trabajo que se acaba de proponer, uno consistirá en implementar una versión extendida del juego de “Piedra, papel o tijera” y otro, la modificación del sistema de sincronización entre el consumidor y los productores de forma que se evite la espera activa. Si se implementa con éxito este último, se podrá subir la nota de la entrega hasta 1 punto.

[4] `rand` genera aleatoriamente números enteros entre 0 y una constante llamada `RAND_MAX`, por lo que el número generado debe ser escalado.

5.1 VERSIÓN EXTENDIDA DEL JUEGO

Existe una versión extendida del juego inventada por Sam Kass y Karen Bryla[5], que consiste en añadir dos opciones más con sus correspondientes reglas. Como curiosidad, esta versión del juego se popularizó gracias a la serie de TV "Big Bang Theory". Las opciones nuevas serían, "lagarto" y "Spock" (el personaje de "Star Trek"). Con respecto a las reglas, quedarían como sigue:

- Las tijeras *cortan* el papel (ganan las tijeras).
- El papel *envuelve* la piedra (gana el papel).
- La piedra *aplasta* al lagarto (gana la piedra).
- El lagarto *envenena* a Spock (gana el lagarto).
- Spock *rompe* las tijeras (gana Spock).
- Las tijeras *decapitan* al lagarto (ganan las tijeras).
- El lagarto *come* el papel (gana el lagarto).
- El papel *refuta* a Spock (gana el papel).
- Spock *vaporiza* la piedra (gana Spock).
- La piedra *aplasta* las tijeras (gana la piedra).

Se propone como ejercicio adicional incorporar lo necesario para que la implementación funcione con las nuevas opciones y reglas.

5.2 SINCRONIZACIÓN SIN ESPERA ACTIVA

Tal como se explicaba en la Sección 4.1, la implementación que se propone utiliza espera activa en lugar de la función pause con objeto de sincronizar el consumidor con los productores. La implementación por medio de `pause` sería incorrecta debido a que existe la posibilidad de que dicha función se llame después de que la señal `SIGUSR1` haya sido recibida y atendida, lo que desembocaría en un bloqueo del productor.

A pesar de que, como se explicaba más arriba, la espera activa no tiene un impacto demasiado negativo en esta implementación, es algo que debería evitarse siempre que sea posible. Se propone como ejercicio adicional, proporcionar una implementación de la sincronización sin espera activa. Dicha implementación se basará en lo siguiente:

- Existe la posibilidad de bloquear señales, de manera que éstas quedan en espera para ser atendidas. El bloqueo se puede llevar a cabo de forma específica por cada señal. Cuando las señales se desbloquean, se atienden en el orden en que fueron recibidas. A fin de bloquear señales, podemos hacer uso de la función `sigprocmask`.
- La función `sigprocmask` trabaja con variables que representan conjuntos de señales que están bloqueadas. A estos conjuntos de señales se les llama máscaras, o en inglés masks. Las máscaras de señales se representan con un tipo específico llamado `sigset_t`. Entender el concepto de máscara de señales es fundamental en la nueva implementación que se propone.
- Para evitar el problema que se ha mencionado con `pause`, es necesario añadir la señal `SIGUSR1` a la máscara de señales bloqueadas mediante `sigprocmask` antes de que la señal pueda enviarse. Por tanto, debería usarse `sigprocmask` dentro de la función `main` antes siquiera de crear los productores mediante `fork`. Se recomienda guardar, tanto la máscara inicial que había antes de llamar a sigprocmask como la nueva máscara que también incluye `SIGUSR1` en dos variables globales.
- En el productor debe quedar bloqueada `SIGUSR1`, mientras que en el consumidor podemos desbloquearla (nuevamente usando `sigprocmask`) antes de ejecutar el resto

[5] http://www.samkass.com/theories/RPSSL.html

del código que ya habíamos implementado (en realidad, no sería estrictamente necesario desbloquearla porque el consumidor no recibirá estas señales, sólo las enviará, no obstante, el código resulta más elegante si se desbloquea).

- Una vez hecho lo anterior, podemos dejar vacío el código de la función que trata la señal `SIGUSR1`, ya que no usaremos espera activa. Por otro lado, el bucle que implementa esta espera activa en el productor ha de sustituirse por una llamada a la función `sigsuspend`. La función `sigsuspend` ejecuta tres acciones. En primer lugar, establece temporalmente una máscara indicando qué señales están bloqueadas. Una vez establecida la nueva máscara se ejecuta internamente la función `pause`, lo que provoca que, si como resultado del establecimiento de la nueva máscara se desbloquean señales que estaban pendientes de ser atendidas, el proceso detendrá su ejecución hasta que se atiendan todas. En tercer y último lugar, la anterior máscara de señales se reestablece y `sigsuspend` termina.
- En nuestro caso, `sigsuspend` ha de establecer una máscara de señales que no bloquea `SIGUSR1` (se puede usar la máscara antigua que devuelve la primera llamada a `sigprocmask`). Esto permite que `sigsuspend` ejecute internamente pause para esperar a que se capture `SIGUSR1` sin ningún peligro de que esta señal ya se hubiera recibido antes, ya que ha permanecido todo el tiempo bloqueado. Además, después de ejecutar pause internamente, la máscara anterior se restablece, por lo que `SIGUSR1` vuelve a estar bloqueada.

Como ayuda adicional para la nueva implementación, son útiles otras dos funciones que se relacionan con el trabajo con máscaras de señales:

- Es posible crear una máscara de señales vacía mediante la función `sigemptyset`.
- Dada una máscara de señales, podemos añadir una nueva señal mediante la función `sigaddset`.

Se recomienda hacer uso del comando `man` para visualizar información sobre las distintas funciones que serán necesarias para la nueva implementación.

6 ENTREGA Y EVALUACIÓN

La entrega consiste en dos partes: **el código fuente** (80% de puntuación) y el **informe** (20% de puntuación).

6.1 CÓDIGO FUENTE

Se pide entregar un único fichero en C implementando la funcionalidad descrita en la Sección 3. Una vez compilado el fichero de código fuente, el programa resultante tendrá tres argumentos: los nombres de los ficheros de opciones (`opciones.csv`) y combinaciones ganadoras (`reglas.csv`) así como el número de turnos, N, que se jugarán.

```
$ ./practica3 <nombre_fichero_opciones.csv> <nombre_fichero_reglas.csv> <valor_N>
```

Si se completa el ejercicio adicional consistente en la implementación sin espera activa, deberá adjuntarse en un fichero aparte (ver formato de la entrega en la Sección 6.3).

6.2 INFORME

El informe a entregar debe prepararse en **formato PDF o equivalente** (no se admiten formatos como odt, docx, etc.). Una forma adecuada de estructurar el informe para esta práctica sería definir tres secciones:

1. **Introducción**: se debe describir brevemente el problema que se ha solucionado (es decir, un resumen de lo que propone esta práctica).
2. **Implementación y pruebas realizadas**: por un lado, de pide describir cómo se ha estructurado el código, enumerando las distintas funciones y proporcionando una breve descripción del cometido de cada una de ellas. También se puede destacar la definición de alguna variable que se considere importante. Por otro lado, también se han de mostrar las pruebas que se han realizado para asegurar el buen funcionamiento del código.
3. **Conclusiones**: se debe hacer una breve reflexión sobre lo aprendido. De manera opcional, también se puede usar esta sección para aportar una reflexión personal sobre la práctica, incluyendo sugerencias sobre aspectos que se podían haber incluido a fin de hacerla más provechosa.

En el caso de que se desee incluir capturas de pantalla en lugar de incluir los resultados de los experimentos en formato texto, es necesario asegurarse de que el texto de la captura se puede leer bien (es decir, que tenga un tamaño similar al resto del texto del documento) y que todas las capturas sean uniformes (es decir, que todas las capturas tengan el mismo tamaño de texto).

El documento debe tener una longitud máxima de 4 páginas (sin incluir índice ni la portada). El documento se evaluará con los siguientes pesos: pruebas realizadas y comentarios asociados, un 60 %; escritura sin faltas de ortografía y/o expresión, un 20 %; paginación del documento hecha de forma limpia y uniforme, 20 %.

6.3 FORMATO DE ENTREGA

Cada grupo debe subir **un único archivo ZIP** a la tarea asociada a esta práctica en el campus virtual con el nombre **P3_GrupoXX_nombre1_apellido1_nombre2_apellido2.zip**, y debe incluir lo siguiente:

- Fichero `practica3.c` implementando el sistema descrito en este documento utilizando espera activa para la sincronización.
- Informe de la práctica en formato `pdf`.
- Fichero `practica3_sigsuspend.c` con la implementación del sistema sin usar espera activa (**OPCIONAL**).
- Ficheros adicionales que puedan ser necesarios en relación a los ejercicios optativos.