# 1 Ejercicio resuelto de pipeline

Se desea informar del tiempo invertido en ejecutar las órdenes **ls | wc -l**. Para ello se escribe el siguiente programa. Sin embargo, no informa de forma correcta. Modifícalo para que lo haga sin cambiar el número de procesos que se están generando.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  #include <sys/types.h>  #include <sys/wait.h>  int main (int argc, char \*argv[]) {  int tubo[2];  time\_t ini, fin;  pipe(tubo);  if (fork() == 0) {  if (fork() == 0) {  dup2(tubo[1], STDOUT\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("ls", "ls", NULL);  } else {  dup2(tubo[0], STDIN\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("wc", "wc", "-l", NULL);  }  }  else {  printf("Tiempo invertido: %ld segundos\n", fin - ini);  }  return 0;  } |

## 1.1 Análisis del código

Línea 1: stdio.h - Standard Input-Output Header" (Cabecera estándar E/S).

Línea 2: stdlib.h - Standard Library o biblioteca estándar).

Línea 3: unistd.h - UNIX Standard (Acceso a la API del sistema operativo POSIX).

Línea 4: time.h - Funciones para manipular y formatear la fecha y hora del sistema.

Línea 5: sys/types.h - Definiciones de tipos especiales (procesos e hilos).

Línea 6: sys/wait.h – Define las constantes simbólicas para usar con waitpid(), la espera para que un proceso hijo se detenga o finalice.

Línea 8: Definición del método main.

Línea 9: Definición de un **arreglo entero** de dos posiciones. El primer elemento del arreglo pasado como parámetro es utilizado para **leer los datos desde el pipe (tubería)**. El segundo elemento del arreglo se usa para **escribir datos a la tubería**.

Línea 10: Definición de las **variables** ini y fin de tipo time\_t (almacenamiento de valores de tiempo del sistema). Se registra la hora de inicio en la variable ini.

Línea 11: pipe() es una función del sistema Linux, usada para abrir los descriptores de los archivos, usados para **comunicarse entre diferentes procesos Linux** (Comunicación entre Procesos). Esta función está definida en la biblioteca unistd.h.

Línea 12: La función fork() se utiliza para crear un nuevo proceso, denominado proceso hijo, el cual se **ejecuta concurrentemente** con el proceso que hace la llamada al proceso fork() (proceso padre). Después de que un nuevo proceso hijo se crea, ambos procesos ejecutarán la próxima instrucción siguiente a la llamada del sistema fork(). El proceso hijo utiliza el mismo PC (Contador de Programa), los mismos registros de la CPU, abre los mismos archivos que son utilizados por el proceso padre. Si la **función retorna cero**, indica que se creó un **nuevo proceso hijo**.

Línea 13: **Creación** de otro proceso.

Línea 14: La función dup2(), que pertenece a la biblioteca unistd.h, crea una copia de un descriptor de archivo existente. En Linux hay **tres tipos de descriptores estándar de archivos**: stdin (entrada desde la terminal por defecto), stdout (imprimir algo a la terminal por defecto ), stderr (imprimir mensajes de error a la terminal por defecto). STDOUT\_FILENO mantiene el valor de 1, el cual es el valor del descriptor de archivos de stdout.

Línea 15: Cierra el descriptor del **archivo de lectura**.

Línea 16: Cierra el descriptor del **archivo de escritura**.

Línea 17: Reemplaza la imagen del proceso actual con una imagen de un nuevo proceso especificado por el archivo. Para este caso se **ejecuta el comando** lssin parámetros.

Línea 18: Cuando finalice el proceso **ls**.

Línea 19: En este caso nos interesa recibir los datos del proceso ls, que con dup2() se había generado como salida, para este caso utilizamos dup2() como entrada para el comando wc -l. Por esta razón. Utilizamos STDIN\_FILENO que mantiene el valor de 0, equivalente a stdin.

Línea 20: Cierra el descriptor del archivo de lectura.

Línea 21: Cierra el descriptor del archivo de escritura.

Línea 22: Ejecutar el comando wc (word count), que cuenta caracteres, palabras o líneas, y se especifica el parámetro (-l) para contar las líneas que recibimos de los datos como entrada para el comando wc, producidos como salida por el comando ls.

Línea 25: Cuando finalice el proceso wc.

Línea 26: Imprime la diferencia de tiempo entre las variables fin e ini.

## 1.2 Ejecución del programa

Verifique los resultados de la instrucción de comandos IPC y la ejecución del programa en C.

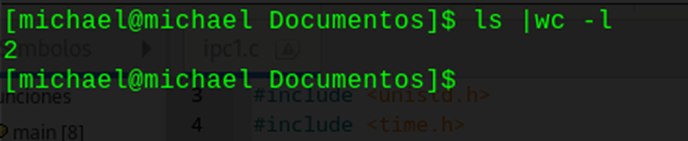
(base) [uptc@fedora c]$ ls | wc -l

(base) [uptc@fedora c]$ gcc ipc1\_ok.c -o ipc1\_ok

(base) [uptc@fedora c]$ ./ipc1\_ok

Veremos la ejecución de los comandos IPC y la ejecución del programa que aun NO está corregido a continuación:

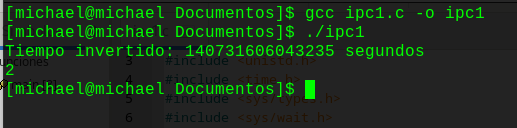
**Figura. Salida del comando ls | wc -l**



*Nota. autoría propia*

En la salida del programa no corregido podemos ver que nos muestra un tiempo invertido mucho mayor de lo que realmente fue, como lo podemos ver en la siguiente figura:

**Figura. Salida del programa aún no corregido**



*Nota. autoría propia*

Por lo cual, a continuación se presenta la respectiva solucion:

## 1.3 Análisis de resultados

- Vemos que pipe se ejecuta antes de crear el proceso hijo con fork(), por lo tanto, es necesario desplazar esta instrucción después del primer if.

- En el último else, el de cierre de los dos procesos, vemos que se intenta imprimir la diferencia de tiempos, pero aún no se ha obtenido el tiempo de finalización de la variable fin. Para su obtención es necesario esperar a finalizar los procesos y ahí sí obtener este tiempo, por lo tanto se necesitan de dos instrucciones:

wait() que bloquea al proceso que hizo la llamada hasta que uno de sus procesos hijos salga o sea recibida una señal. Después de que el proceso hijo termina, el padre continúa su ejecución después de la instrucción de llamada al sistema wait.

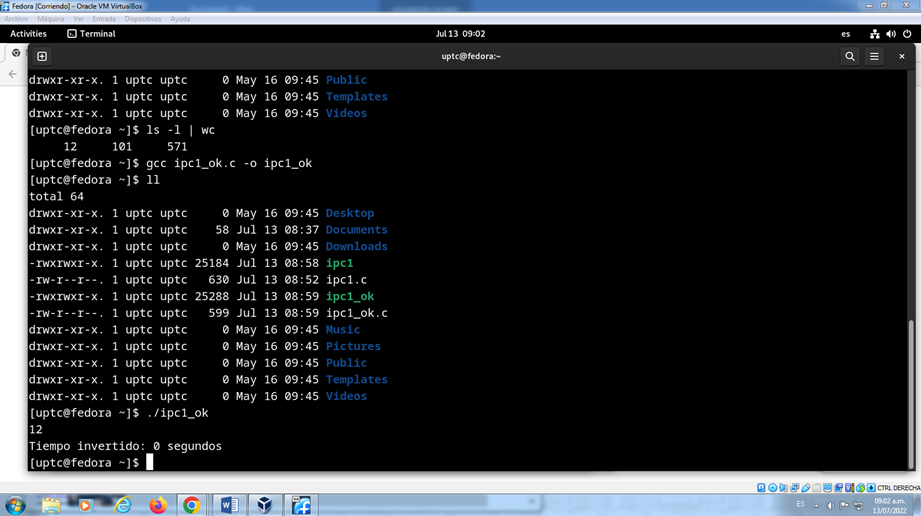
fin = time (NULL)que registra la hora de finalización.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  #include <sys/types.h>  #include <sys/wait.h>  int main (int argc, char \*argv[]) {  int tubo[2];  time\_t ini = time(NULL), fin;  if (fork() == 0) {  pipe(tubo);  if (fork() == 0) {  dup2(tubo[1], STDOUT\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("ls", "ls", NULL);  } else {  dup2(tubo[0], STDIN\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("wc", "wc", "-l", NULL);  }  }  else {  wait(NULL);  fin = time(NULL);  printf("Tiempo invertido: %ld segundos\n", fin - ini);  }  return 0;  } |

Para comprobar y comparar los resultados obtenidos con el script YA corregido, ejecutamos tanto el comando “ ls | wc - l “, como el respectivo script en nuestras máquinas virtuales, obteniendo el siguiente resultado para el comando equivalente, el cual lo podemos ver en la siguiente figura:

Donde, realizamos la compilamos y ejecutamos el script obteniendo el resultado visto en la siguiente figura:

**Figura. Salida del Script ya corregido**



Nota: autoría propia

Donde notamos que además de la salida similar que nos debería mostrar con el comando ls | wc -l, también este script nos permite visualizar el tiempo invertido en segundos del programa corregido

**Nota:** Se aclara que la imagen (**Salida del Script ya corregido),** es tomada de la actividad realizada en clase y por ello la cantidad de archivos y directorios cambia con respecto a la imagen tomada desde nuestras propias máquinas virtuales, pero, lo que se quiere mostrar es que el tiempo invertido del script corregido muestra el tiempo en segundo real que se buscaba corregir a comparación del que aún no estaba corregido. Ya que la cantidad de archivos existentes los contaba correctamente.

# 2. Ejercicio pipeline por resolver

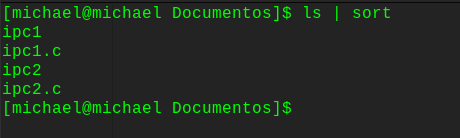
Observa el siguiente fragmento de código que trata de realizar lo mismo que si un usuario escribiera ls | sort en la línea de comandos de un sistema Linux. Indica qué problemas se producen, por qué y cómo resolverlos.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  #include <sys/types.h>  #include <sys/wait.h>  int main (int argc, char \*argv[]) {  int tubo[2];  pipe(tubo);  if (fork() != 0) {  dup2(tubo[1], STDIN\_FILENO);  execlp("sort", "sort", NULL);  close(tubo[0]);  close(tubo[1]);  } else {  dup2(tubo[0], STDOUT\_FILENO);  close(tubo[1]);  close(tubo[0]);  execlp("ls", "ls", NULL);  }  return 0;  } |

**Salida del comando ls | sort:**

A continuación veremos lo que obtuvimos al ejecutar el comando equivalente al codigo que queremos realizar, en la siguiente figura:

**Figura. Resultados obtenidos con el comando “ls | sort”**



*Nota: autoría propia*

Como podemos ver, la salida de este comando en Linux arroja un listado archivos, ordenados alfabéticamente, esto se debe a que el proceso “ls” lista los archivos contenido en el directorio, lista la cual sirve como entrada para el proceso “sort”, el cual se encarga de ordenar dicha lista alfabéticamente

**Solución del problema:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/types.h>

#include <sys/wait.h>

int main (int argc, char \*argv[]) {

int tubo[2];

pipe(tubo);

if (fork() != 0) {

dup2(tubo[0], STDIN\_FILENO);

close(tubo[0]);

close(tubo[1]);

execlp("sort", "sort", NULL);

} else {

dup2(tubo[1], STDOUT\_FILENO);

close(tubo[1]);

close(tubo[0]);

execlp("ls", "ls", NULL);

}

return 0;

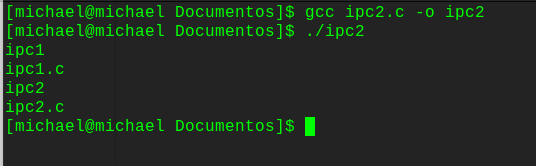
}

Para solucionar este error, se realizaron dos correcciones. Primero, dentro del primer condicional, como parámetro de dup2, se envió la posición 0 de la arreglo de enteros tubo, en lugar de la posición 1, y así mismo se corrigió en el else del condicional, en el que se cambio el parámetro recibido por la operación dup2, pero en este caso se le envía la posición 1 del arreglo tubo

Por último se bajo dos líneas la instrucciones excelp(“sort”, “sort, NULL) y excelp(“ls”, “ls”, NULL), para la ejecución pudiera terminar de manera correcta y no se quedará esperando una salida

La salida del script corregido, la vemos en la siguiente figura:

**Figura. Salida del script corregido**



*Nota: autoría propia*

En la anterior imagen podemos comprobar el correcto funcionamiento del código, ya que arroja la misma salida que el comando “ls | sort”.

# 3. Segundo Ejercicio pipeline por resolver

Al ejecutar el siguiente programa, el proceso no termina. Explica por qué. Da una solución que no cambie el número de procesos que se generan.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  #include <sys/types.h>  #include <sys/wait.h>  int main(int argc, char \*argv[]) {  int tubo[2];  pipe(tubo);  if (fork() == 0) {  if (fork() == 0) {  dup2 (tubo[1], STDOUT\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("ls", "ls", NULL);  } else {  dup2 (tubo[0], STDIN\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("wc", "wc", "-l", NULL);  }  } else {  wait(NULL);  printf ("Fin del proceso\n");  }  } |

## 3.1 Desarrollo del Código con Errores

Vemos el código en una de nuestras máquinas virtuales en el respectivo editor de texto que se tiene instalado en la siguiente figura:

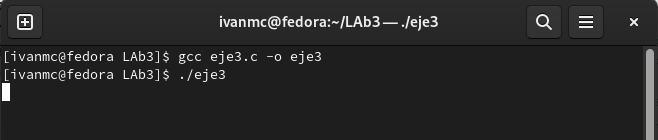
**Figura. Código realizado en una de nuestras máquinas virtuales**

****

*Nota: autoría propia*

En primer lugar, comprobamos que realmente sucedía lo que se nos menciona en el enunciado con el programa, es decir, que el proceso no termina. Por lo cual pasamos a ejecutar el programa obteniendo el resultado que se puede ver en la siguiente figura:

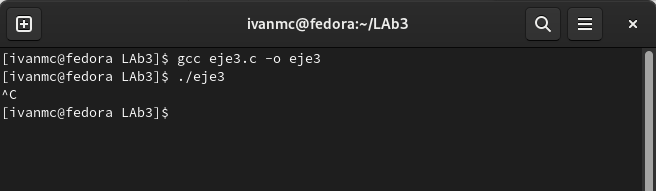
**Figura. Ejecución del programa sin corregir (donde vemos que no termina).**



*Nota: autoría propia*

Por lo cual, tenemos que matar el proceso nosotros mismos con la combinación de las teclas: “control” + “c”, como vemos en la siguiente imagen:

**Figura. Matando el proceso por nuestra cuenta**



*Nota: autoría propia*

## 3.2 Análisis del Código

**Inclusión de librerías necesarias para ejecutar el programa**

**Línea 1:** stdio.h - Standard Input-Output Header" (Cabecera estándar E/S).

**Línea 2:** stdlib.h - Standard Library o biblioteca estándar).

**Línea 3:** unistd.h - UNIX Standard (Acceso a la API del sistema operativo POSIX).

**Línea 4:** time.h - Funciones para manipular y formatear la fecha y hora del sistema.

**Línea 5:** sys/types.h - Definiciones de tipos especiales (procesos e hilos).

**Línea 6:** sys/wait.h – Define las constantes simbólicas para usar con waitpid(), la espera para que un proceso hijo se detenga o finalice.

**Código realizado por el programador**

**Línea 8:** Definición del método main.

**Línea 9:** Definición de un **arreglo entero** de dos posiciones. El primer elemento del arreglo pasado como parámetro es utilizado para **leer los datos desde el pipe (tubería)**. El segundo elemento del arreglo se usa para **escribir datos a la tubería**.

**Línea 10:** pipe() es una función del sistema Linux, usada para abrir los descriptores de los archivos, usados para **comunicarse entre diferentes procesos Linux** (Comunicación entre Procesos). Esta función está definida en la biblioteca unistd.h.

**Línea 11:** La función fork() se utiliza para crear un nuevo proceso, denominado proceso hijo, el cual se **ejecuta concurrentemente** con el proceso que hace la llamada al proceso fork() (proceso padre). Después de que un nuevo proceso hijo se crea, ambos procesos ejecutarán la siguiente instrucción siguiente a la llamada del sistema fork(). El proceso hijo utiliza el mismo PC (Contador de Programa), los mismos registros de la CPU, abre los mismos archivos que son utilizados por el proceso padre. Si la **función retorna cero**, indica que se creó un **nuevo proceso hijo**.

**Línea 12:** **Creación** de otro proceso.

**Línea 13:** La función dup2(), que pertenece a la biblioteca unistd.h, crea una copia de un descriptor de archivo existente. En Linux hay **tres tipos de descriptores estándar de archivos**: stdin (entrada desde la terminal por defecto), stdout (imprimir algo a la terminal por defecto ), stderr (imprimir mensajes de error a la terminal por defecto). STDOUT\_FILENO mantiene el valor de 1, el cual es el valor del descriptor de archivos de stdout.

**Línea 14:** Cierra el descriptor del **archivo de lectura**.

**Línea 15:**  Cierra el descriptor del **archivo de escritura**.

**Línea 16:** Reemplaza la imagen del proceso actual con una imagen de un nuevo proceso especificado por el archivo. Para este caso se **ejecuta el comando** lssin parámetros.

**Línea 17:** Cuando finalice el proceso **ls**.

**Línea 18:** En este caso nos interesa recibir los datos del proceso ls, que con dup2() se había generado como salida, para este caso utilizamos dup2() como entrada para el comando wc -l. Por esta razón. Utilizamos STDIN\_FILENO que mantiene el valor de 0, equivalente a stdin.

**Línea 19:** Cierra el descriptor del archivo de lectura.

**Línea 20:** Cierra el descriptor del archivo de escritura.

**Línea 21:** Ejecutar el comando wc (word count), que cuenta caracteres, palabras o líneas, y se especifica el parámetro (-l) para contar las líneas que recibimos de los datos como entrada para el comando wc, producidos como salida por el comando ls.

**Línea 23:** Cuando finalice el proceso wc.

**Línea 24:** wait() que bloquea al proceso que hizo la llamada hasta que uno de sus procesos hijos salga o sea recibida una señal

**Línea 25:** Imprime un mensaje que dice: Fin de proces

## 3.1 Desarrollo del Código sin Errores

El código corregido es el siguiente:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  #include <sys/types.h>  #include <sys/wait.h>  int main(int argc, char \*argv[]) {  int tubo[2];  if (fork() == 0) {  pipe(tubo);  if (fork() == 0) {  dup2 (tubo[1], STDOUT\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("ls", "ls", NULL);  } else {  dup2 (tubo[0], STDIN\_FILENO);  close(tubo[0]);  close(tubo[1]);  execlp("wc", "wc", "-l", NULL);  }  } else {  wait(NULL);  printf ("Fin del proceso\n");  }  return 0;  } |
|  |  |

La descripción de la corrección realizada es la siguiente.

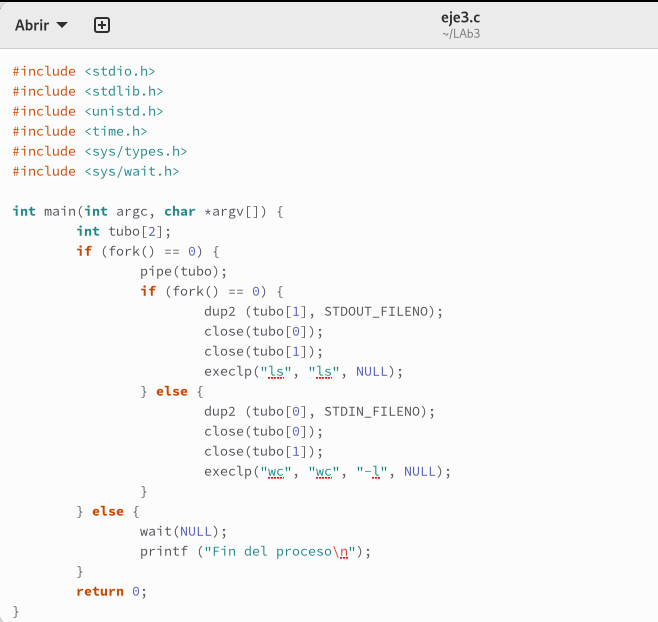
Notamos que el problema ocurre, es que el proceso inicial que se crea, es decir, el primer fork, tiene acceso a la **tubería**, y no lograr cerrar el descriptor de escritura, por lo cual, el proceso cambia al comando “wc”, el cual debería realizar la lectura de la tubería se bloquea al vaciarlo, y es el comando “ls” el que está realizando la lectura de la tubería.

Por lo cual, la solución que pudimos dar es bastante sencilla, y es que el proceso inicial que se crea no va a hacer uso del **pipe**, y pasamos el **pipe** después del primer condicional, para que así se le pueda dar acceso a los demás procesos y de igual forma, evitando que el proceso incial pueda acceder a este pipe.

Así logramos que el proceso no termine y realice el respectivo conteo de los archivos que se encuentran en la carpeta, y para corroborar el resultado, ejecutamos el comando “ ls | wc - l” para comparar los resultados obtenidos por este comando y el programa corregido, obteniendo resultados iguales.

Además, también notamos que a este programa le faltaba retornar algo, por lo cual, al finalizar el código, también se le añade “return 0”. Veremos el código corregido en la siguiente figura:

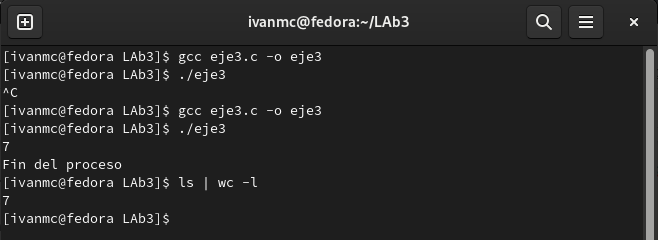
**Figura. Corrección del código del programa realizado**



*Nota: autoría propia*

Y a continuación veremos los resultados obtenidos tanto con la ejecución del programa corregido y el comando “ ls | wc -l “, en la siguiente figura:

**Figura. Resultados obtenidos tanto con la ejecución del programa como con el comando equivalente.**



*Nota: autoría propia*

Así es como concluimos este punto del laboratorio.

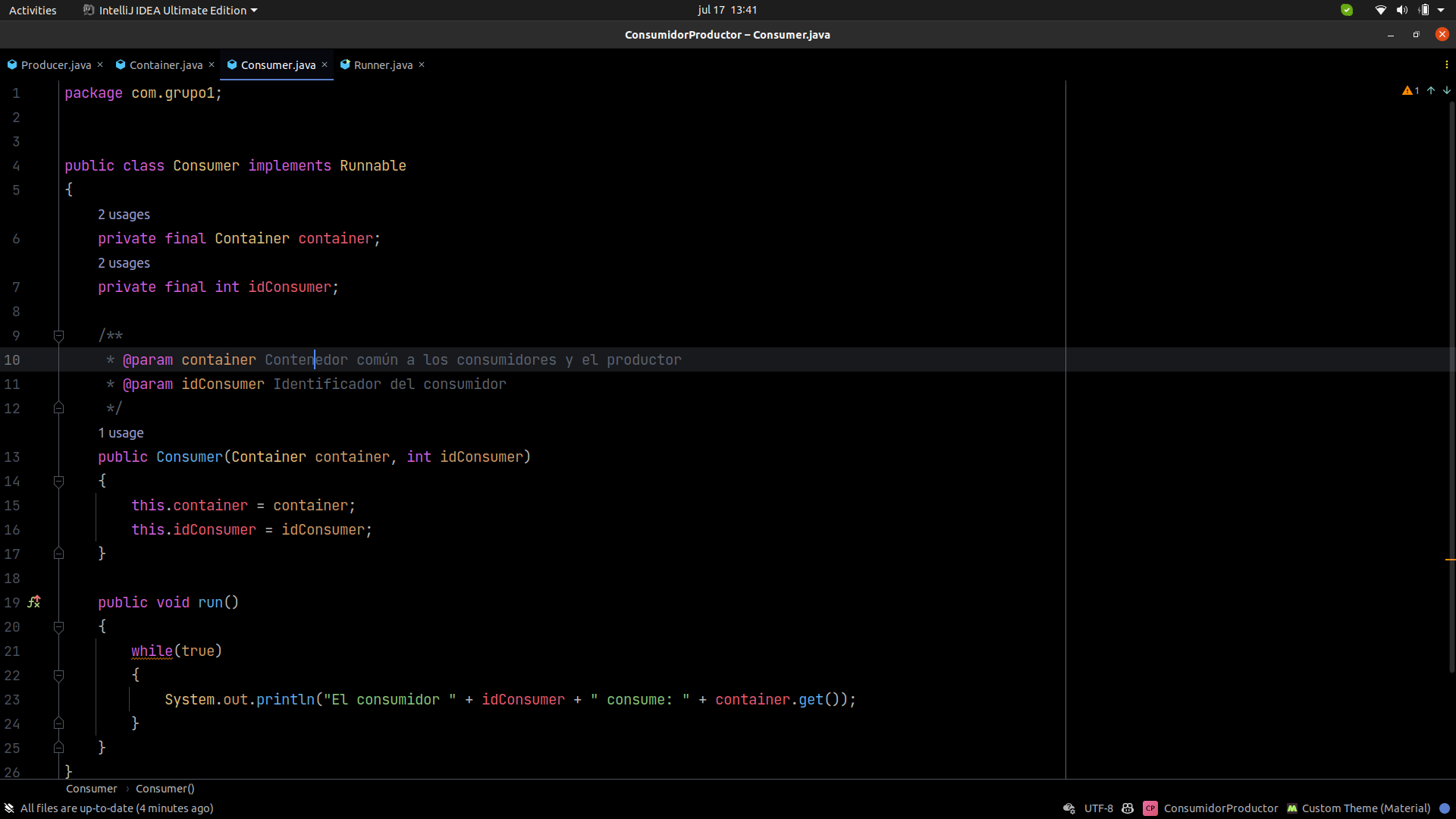
# 4. Algoritmo de productor y consumidor

En el lenguaje de programación de su preferencia implemente la solución del algoritmo de productor y consumidor utilizando monitores.

Para implementar el algoritmo de productor y consumidor usamos el lenguaje de programación **Java**, donde se crearon 4 clases: **Consumer, Producer, Container, Runner.**

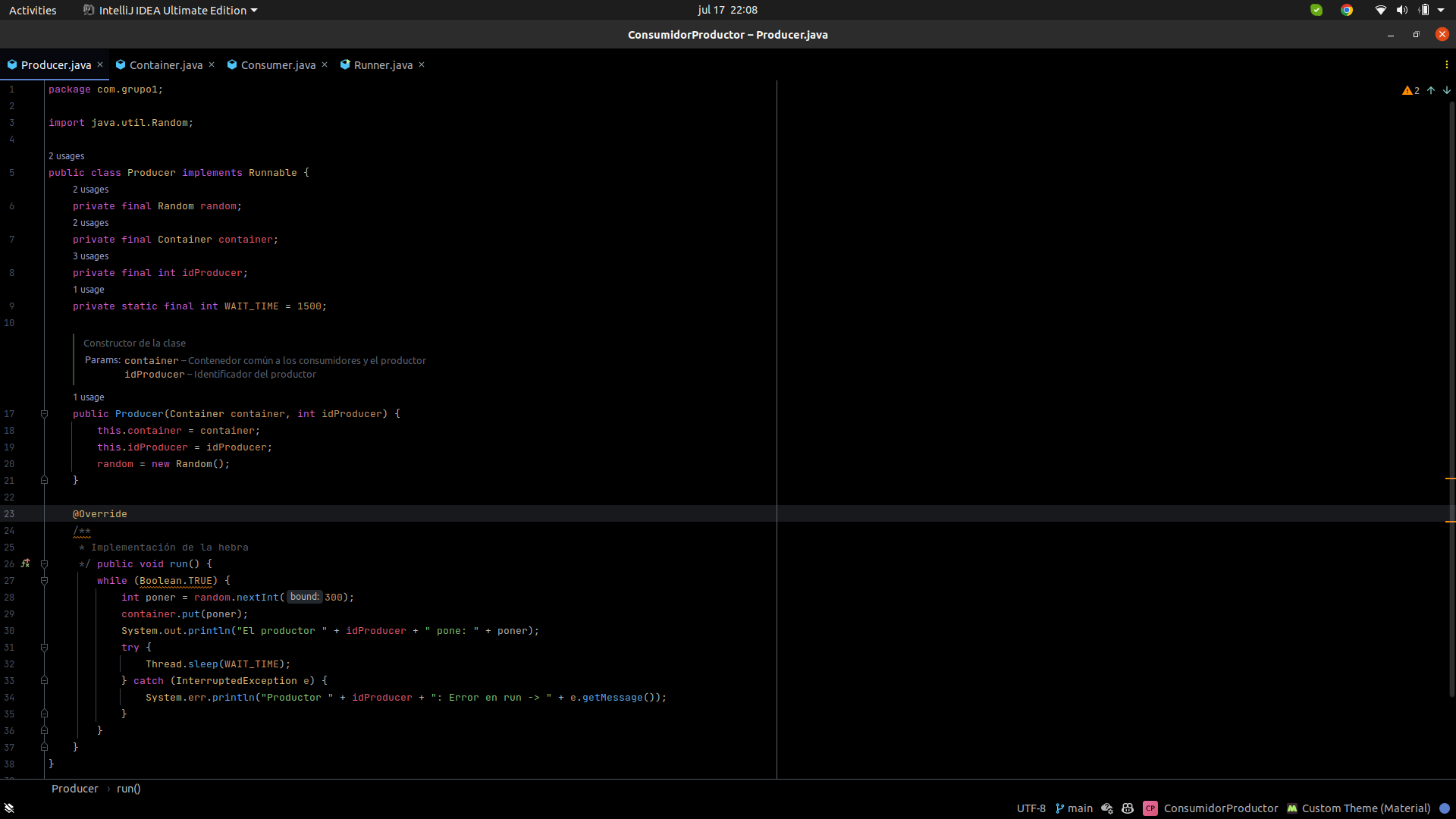
La clase Runner, es la encargada de iniciar toda la ejecución del programa y este se ejecutará de forma automática y sin fin, a menos que el usuario haga una interrupción, cabe resaltar que el programa tiene una salida en consola y se trata de consumir números.

La clase Consumer es la siguiente:



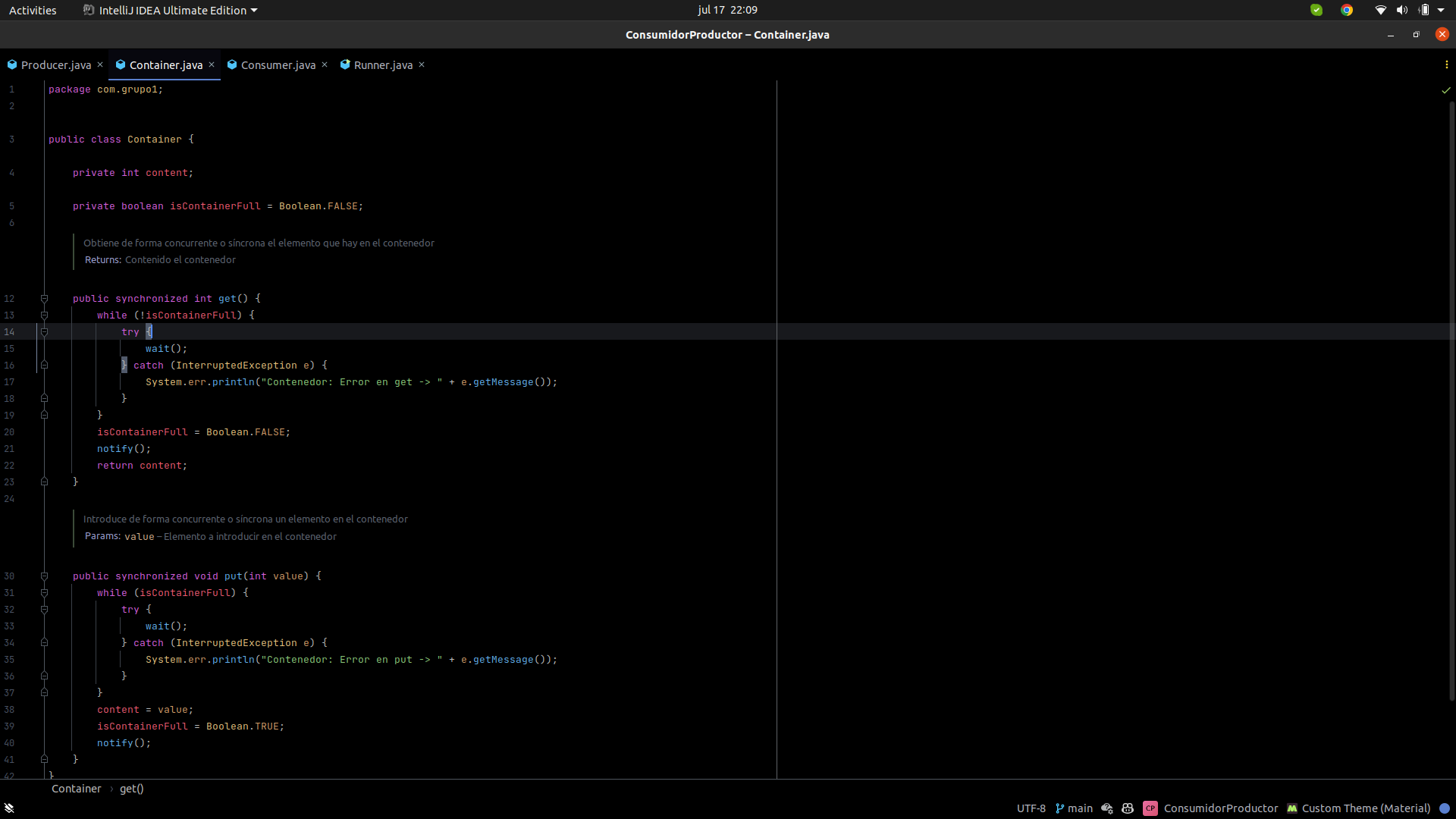
Aquí encontramos que para crear un consumidor necesitamos el contenedor y un id para este consumidor.

Después tenemos La clase Producer, que tiene la lógica de lo que hace el productor como podemos ver en el método run.

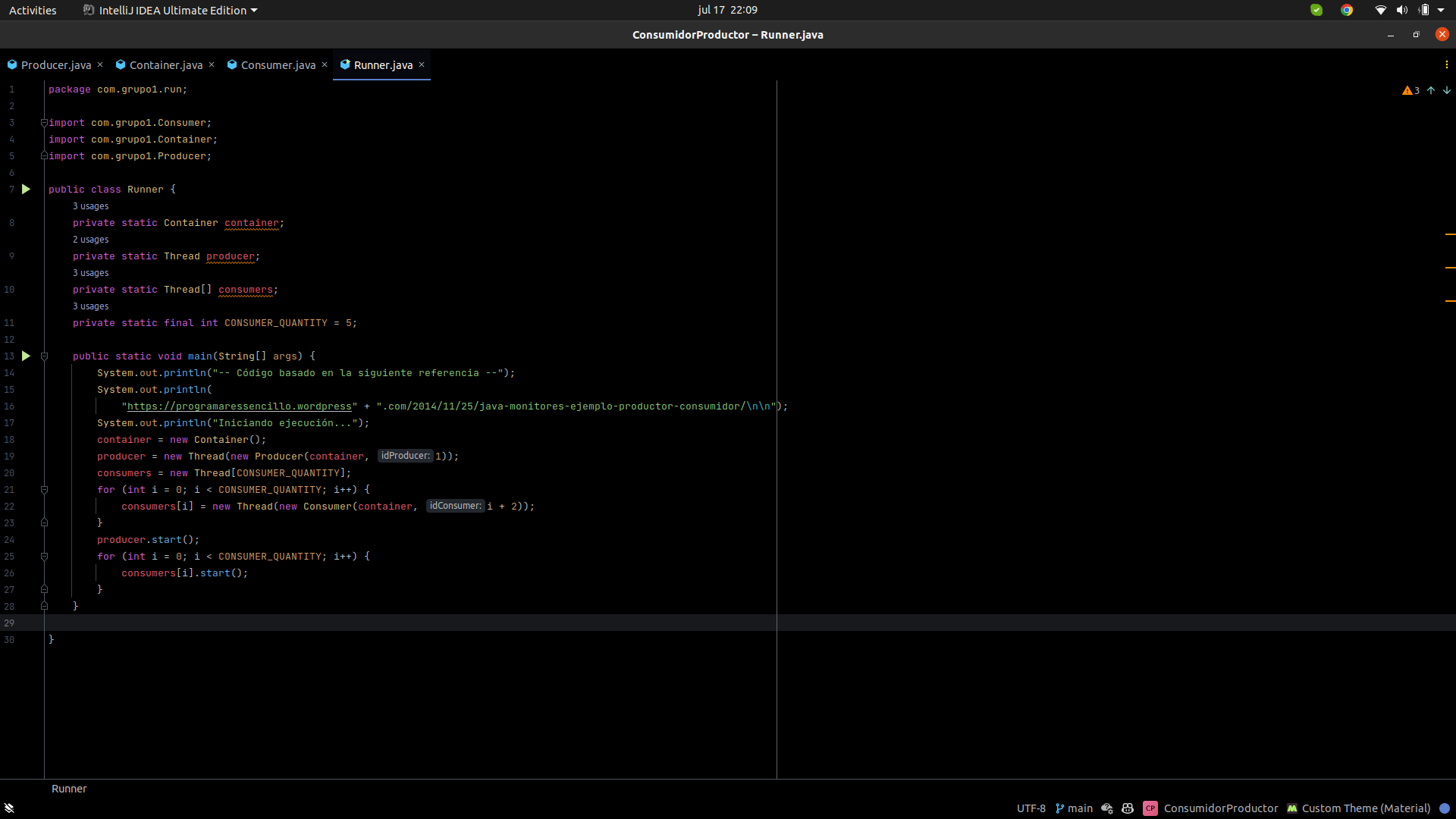


Podemos ver que estamos usando números para identificar la interacción entre el consumidor y el productor.

Con esto tenemos la clase container que es quien actúa como monitor en este caso.



Y por ultimo tenemos la clase Runner, encargada de correr el programa al mismo tiempo que inicializa algunos valores como el numero de consumidores o el productor.



# Conclusiones

* Los Pipe comunican procesos que tienen que estar relacionados, En un mecanismo IPC, esta relación puede ser padre a hijo o hijo a hijo. como observamos en los ejercicios, que la organización de la función pipe() para usar para comunicarse entre diferentes procesos y la función fork() utilizada para crear un nuevo proceso, denominado proceso hijo influye en los resultados del código.
* Notamos que al realizar las respectivas correcciones en cada uno de los scripts, los errores encontrados comúnmente se basaban en el orden en que se ejecutaban las instrucciones por lo cual debíamos realizar el respectivo análisis de cuando se ejecutaba cierto pipe y si esto no dejaba que entrara a los demás condicionales, o también en cuestión del orden en el cual se ejecutaban los comandos

# Referencias

[1] A. S. Tanenbaum, *Sistemas Operativos Modernos*. Prentice Hall, 2015