TEMA1 . PROGRAMACIÓN DE PROCESOS

Objetivos

- Comprender de los conceptos básicos del funcionamiento de los sistemas en lo relativo a la ejecución de diferentes programas.
- Comprender el concepto de concurrencia y cómo el sistema puede proporcionar multiprogramación al usuario.
- Entender las políticas de planificación del sistema para proporcionar multiprogramación y multitarea.
- Familiarizarse con la programación de procesos entendiendo sus principios y formas de aplicación.

1.1 CONCEPTOS BÁSICOS

- **Aplicación:** Es un tipo de programa informático (software), diseñado como herramienta ante un problema.
- **Programa**: Es el conjunto de instrucciones que ejecutadas en un ordenador realizarán una tarea.
- **Proceso**: Programa en ejecución. Este concepto no se refiere únicamente al código y a los datos, sino que incluye todo lo necesario para su ejecución:
    - Contador de programa. (en qué instrucción del programa o instante de ejecución se encuentra).
    - Imagen de memoria.
    - Estado del procesador

  Los procesos son entidades independientes, aunque ejecuten el mismo programa. De tal forma, pueden coexistir dos procesos que ejecuten el mismo programa, pero con diferentes datos (es decir, con distintas imágenes de memoria) y en distintos momentos de su ejecución (con diferentes contadores de programa).
- **Ejecutable**: Es un fichero que contiene la información necesaria (el código binario o interpretado) para crear un proceso.
  Tipos:
    - Según el sistema operativo:
        - Windows, utiliza como fichero ejecutable .exe principalmente.
        - Linux, posible cambiarle permisos de ejecución a cualquier fichero mediante el comando chmod.
    - Según el código del ejecutable:
        - Binario. Código compilado obteniendo un fichero en lenguaje máquina. Dependiente del S.O. del equipo.
        - Interpretados. No es un código binario sino de bytecodes o códigos de operación que son interpretados por un intérprete (máquina virtual Java o JRE) que los convierte a lenguaje máquina.
- **Sistema operativo:** Programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador. Entre sus objetivos, se pueden destacar:

- Ejecutar los programas del usuario. Es el encargado de crear los procesos a partir de los ejecutables de los programas y de gestionar su ejecución para evitar errores y mejorar el uso del computador.
- Hacer que el computador sea cómodo de usar. Hace de interfaz entre el usuario y los recursos del ordenador, permitiendo el acceso tanto a ficheros y memoria como a dispositivos hardware. Esta serie de abstracciones permiten al programador acceder a los recursos hardware de forma sencilla
- Utilizar los recursos del computador de forma eficiente. Los recursos del ordenador son compartidos tanto por los programas como por los diferentes usuarios. El sistema operativo es el encargado de repartir los recursos en función de sus políticas a aplicar.

1.2 PROGRAMACIÓN CONCURRENTE

La **computación concurrente** permite la posibilidad de tener en ejecución al mismo tiempo múltiples tareas interactivas. Es decir, permite realizar varias cosas al mismo tiempo, como escuchar música, visualizar la pantalla del ordenador, imprimir documentos, etc.
Dichas tareas se pueden ejecutar en:

- **Un único procesador (multiprogramación)**. Solamente un proceso puede estar en un momento determinado en ejecución. Para poder ir cambiando entre los diferentes procesos, el sistema operativo se encarga de cambiar el proceso en ejecución después de un período corto de tiempo. Esto permite que en un segundo se ejecuten múltiples procesos, creando en el usuario la percepción de que múltiples programas se están ejecutando al mismo tiempo.
  Este concepto se denomina **programación concurrente**. La programación concurrente no mejora el tiempo de ejecución global de los programas ya que se ejecutan intercambiando unos por otros en el procesador. Sin embargo, permite que varios programas parezca que se ejecuten al mismo tiempo.
- **Varios núcleos en un mismo procesador (multitarea)**. La existencia de varios núcleos o cores en un ordenador. Cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo. El sistema operativo, al igual que para un único procesador, se debe encargar de planificar los trabajos que se ejecutan en cada núcleo y cambiar unos por otros para generar multitarea.
  Todos los cores comparten la misma memoria por lo que es posible utilizarlos de forma coordinada mediante lo que se conoce por **programación paralela.**

  ACLARAR: En cada instante sólo se ejecutará un único proceso de forma secuencial, pero en caso de múltiples núcleos ante la ejecución de un proceso este podrá llevar a cabo múltiples instrucciones de forma paralela (multitarea) mediante hilos que utilizan los mismo recursos que el proceso al que pertenecen.

1.3 FUNCIONAMIENTO BÁSICO DEL SISTEMA OPERATIVO

La parte central que realiza la funcionalidad básica del sistema operativo se denomina **kernel**. El kernel es el responsable de gestionar los recursos del ordenador, permitiendo su uso a través de **llamadas al sistema.**

En general, el kernel del sistema funciona en base a interrupciones.
Las llamadas al sistema son la interfaz que proporciona el kernel para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema.

1.4 PROCESOS

A medida que se ejecuta un proceso, dicho proceso pasará por varios estados. El cambio de estado también se producirá por la intervención del sistema operativo.

1.4.1 ESTADO DE UN PROCESO

Los estados de un proceso son:

- **Nuevo**. El proceso está siendo creado a partir del fichero ejecutable.
- **Listo**: El proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
- **En ejecución**: El proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. Si el proceso necesita un recurso, incluyendo la realización de operaciones de entrada/salida (E/S) lo retira del estado de "en ejecución" al estado "Bloqueado".
  Si un proceso en ejecución se ejecuta durante el tiempo máximo permitido por la política del sistema, salta un temporizador que lanza una interrupción. En este último caso, pasa al estado Listo, y el S.O. selecciona otro proceso para que continúe su ejecución.
- **Bloqueado:** El proceso está bloqueado esperando que ocurra algún suceso. Cuando ocurre el evento que lo desbloquea, el proceso no pasa directamente a ejecución sino que tiene que ser planificado de nuevo por el sistema.
- **Terminado:** el proceso ha finalizado su ejecución y libera su imagen de memoria. Para terminar un proceso, el mismo debe llamar al sistema para indicarselo o puede ser el propio sistema el que finalice el proceso mediante una excepción.

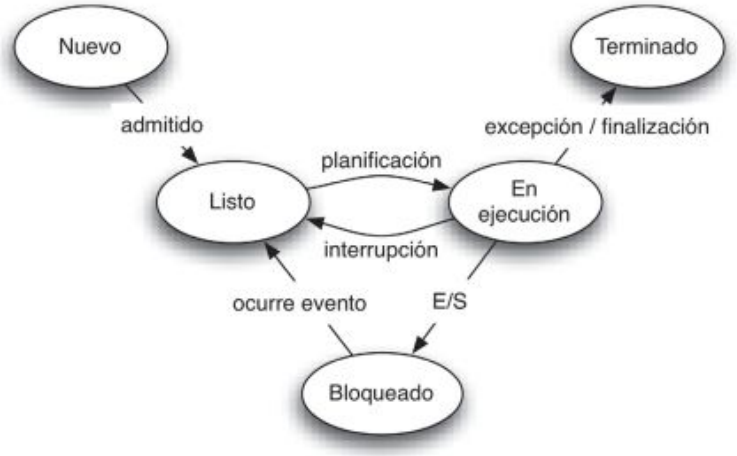

***Figura 1.3.** Estados de un proceso*

1.4.2 COLAS DE PROCESOS

Uno de los objetivos del sistema operativo es la multiprogramación, es decir, admitir varios procesos en memoria para maximizar el uso del procesador. Esto funciona ya que los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas, migrándolos de unas colas a otras:

- **Una cola de procesos** que contiene **todos** los procesos del sistema.
- **Una cola de procesos preparados** que contiene todos los procesos listos esperando para ejecutarse.
- **Varias colas de dispositivo** (una por dispositivo). que contienen los procesos que están a la espera de alguna operación de E/S

Cada una de las colas utiliza el mecanismo FIFO (First In First Out): El primer elemento que se almacena es el primero que sale.

1.4.3 PLANIFICACIÓN DE PROCESOS

Para gestionar las colas de procesos, es necesario un planificador de procesos. El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas. Existen dos tipos de planificación:

- **A largo plazo (Baja frecuencia):** Atiende las solicitudes de los nuevos procesos, tomando la decisión de admitir o no. Controla el grado de multiprogramación (número de procesos en memoria)
- **A corto plazo (Alta frecuencia):** selecciona qué proceso de la cola de procesos preparados pasará a ejecución. Muy alta frecuencia fruto de interrupción de reloj, E/S o llamadas al sistemas. Dada la alta frecuencia de operación los algoritmos deben ser sencillos:
    - **Planificación sin desalojo o cooperativa.** Retirar el proceso en ejecución si dicho proceso se bloquea o termina.
    - **Planificación apropiativa**. La aparición de un proceso más prioritario retira el proceso actual y ejecución del proceso más prioritario.
    - **Tiempo compartido**: Cada cierto tiempo (llamado “cuanto”) se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse.

1.4.4 CAMBIO DE CONTEXTO

Al intercalar diferentes procesos, el sistema operativo debe guardar el contexto del proceso actual y restaurar el contexto del proceso que el planificador a corto plazo ha elegido ejecutar.

Se conoce como contexto a:

- Estado del proceso.
- Estado del procesador: valores de los diferentes registros del procesador.
- Información de gestión de memoria: espacio de memoria reservada para el proceso.

Por lo general, el cambio de contexto es tiempo perdido, ya que el procesador no hace trabajo útil durante ese tiempo. Únicamente es tiempo necesario para permitir la multiprogramación y con ello la concurrencia.

1.5 GESTIÓN DE PROCESOS

1.5.1 ÁRBOL DE PROCESOS

El sistema operativo es el encargado de crear y gestionar los **nuevos procesos**.
Cuando un usuario quiere abrir un programa, el sistema operativo es el responsable de crear y poner en ejecución el proceso correspondiente que lo ejecutará.
Aunque el responsable del proceso de creación es el sistema operativo (único que puede acceder a los recursos del ordenador), el **nuevo proceso se crea siempre por petición de otro proceso**: Un nuevo proceso se produce debido a que un proceso pidió su creación y por lo tanto el proceso en ejecución siempre depende del proceso que lo creó, estableciéndose un vínculo entre ambos.
A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos**.
Para identificar a los procesos, los sistemas operativos suelen utilizar un identificador de proceso (process identifier [PID]) unívoco para cada proceso. La utilización del PID es básica a la hora de gestionar procesos, ya que es la forma que tiene el sistema de referirse a los procesos que gestiona.

1.5.2 OPERACIONES BÁSICAS CON PROCESOS

Dado el árbol de procesos, el proceso creador se denomina padre y el proceso creado se denomina hijo. A su vez, los hijos pueden crear nuevos hijos (**operación "create"**).
Al crearse un nuevo proceso, el proceso padre e hijo se ejecutan concurrentemente y siguen la misma política de planificación del S.O. que vimos hasta ahora como si fueran dos procesos diferentes. Como hemos visto los procesos son independientes y tienen su propia imagen de memoria (espacio de memoria asignada).
La única salvedad es que:

- Si el proceso padre necesita esperar hasta que el hijo termine su ejecución para poder continuar la suya con los resultados obtenidos por el hijo, puede hacerlo mediante la **operación wait.**
- Pueden **compartir recursos para intercambiarse información**. Estos recursos pueden ir desde ficheros hasta zonas de memoria compartida. El sistema operativo solamente interviene a la hora de crear y establecer los permisos de qué procesos pueden acceder a dicha zona.

Al **terminar la ejecución** de un proceso, es necesario avisar al sistema operativo de su terminación para que de esta forma el sistema libere si es posible los recursos que tenga asignados.

- Un proceso hijo puede en la fase de terminación, aprovechar para mandar información respecto a su finalización al proceso padre.

El proceso hijo depende tanto del sistema operativo como del proceso padre que lo creó. Así, el **padre puede terminar la ejecución de un proceso hijo cuando crea conveniente**. Para ello puede utilizar la operación **destroy**.

1.5.2.1 Creación de procesos (operación create)

La clase que representa un proceso en Java es la **clase Process**. Los métodos de **ProcessBuilder.start() y Runtime .exec()** crean un proceso nativo en el sistema operativo y devuelven un objeto de la clase Process que puede ser utilizado para controlarlo.

- **Process ProcessBuilder.start()**: inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso:
  - Ejecuta el comando y los argumentos indicados en el método **command()**.
  - Ejecutándose en el directorio de trabajo especificado por directory().
  - Utilizando las variables de entorno definidas en environment().
- **Process Runtime.exec(String[] cmdarray, String[] envp, File dir)**: ejecuta el comando especificado y argumentos en cmdarray en un proceso hijo independiente con el entorno envp y el directorio de trabajo especificado en dir.

Creación de un proceso utilizando *ProcessBuilder*

```
import java.io.IOException;
import java.util.Arrays;

public class RunProcess {

    public static void main(String[] args) throws IOException {

     if (args.length <= 0) {
         System.err.println("Se necesita un programa a ejecutar");
              System.exit(-1);
     }

    ProcessBuilder pb = new ProcessBuilder(args);
    try {
         Process process = pb.start();
         int retorno = process.waitFor();
         System.out.println("La ejecución de " +
                    Arrays.toString(args) + " devuelve " +  retorno);
     }catch(IOException ex){
          System.err.println("Excepción de E/S!!");
                 System.exit(-1);
    }catch(InterruptedException ex){
         System.err.println("El proceso hijo finalizó
                    de forma incorrecta");
         System.exit(-1);
    }
    }
}
```

1.5.2.2 Terminación de procesos (operación destroy)

Un proceso puede terminar de forma abrupta un proceso hijo que creó. Para ello el proceso padre puede ejecutar la operación **destroy**. Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo subyacente.

Si no se fuerza la finalización de la ejecución del proceso hijo de forma anómala, el proceso hijo realizará su ejecución completa (cuando el hijo realiza la operación exit para finalizar su ejecución).

Creación de un proceso mediante *Runtime* para después destruirlo.

```
import java.io.IOException;

public class RuntimeProcess {

    public static void main(String[] args) {

         if (args.length <= 0) {
                   System.err.println("Se necesita un programa a ejecutar");
                   System.exit(-1);
         }

         Runtime runtime = Runtime.getRuntime();
         try {
              Process process = runtime.exec(args);
              process.destroy();
         }catch(IOException ex){
                   System.err.println("Excepción de E/S!!");
                   System.exit(-1);
         }
         }
}
```

1.6 COMUNICACIÓN DE PROCESOS

Un proceso es un programa en ejecución y, como cualquier programa, recibe información, la transforma y produce resultados. Esta acción se gestiona a través de:

- **La entrada estándar (stdin)**: lugar de donde el proceso lee los datos de entrada que requiere para su ejecución. Normalmente suele ser el teclado, pero podría recibirlos de un fichero, de la tarjeta de red o hasta de otro proceso, entre otros sitios.
- **La salida estándar (stdout)**: el proceso escribe los resultados que obtiene. Normalmente es la pantalla, aunque podría ser, entre otros, la impresora o hasta otro proceso que necesite esos datos como entrada.
    - System.out.println en Java.
- **La salida de error (stderr)**: El proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, por ejemplo un fichero (fichero de logs).

Creación de un proceso mediante *Runtime* para después destruirlo.

```
import java.io.IOException;

public class RuntimeProcess {

    public static void main(String[] args) {

         if (args.length <= 0) {
                   System.err.println("Se necesita un programa a ejecutar");
                   System.exit(-1);
         }

         Runtime runtime = Runtime.getRuntime();
         try {
              Process process = runtime.exec(args);
              process.destroy();
         }catch(IOException ex){
                   System.err.println("Excepción de E/S!!");
                   System.exit(-1);
         }
         }
}
```

En Java, el proceso hijo creado de la clase Process no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente. Todas sus salidas y entradas de información (stdin, stdout y stderr) se redirigen al proceso padre:

- **OutputStream**: flujo de salida del proceso hijo. El stream está conectado por un pipe a la entrada estándar (stdin) del proceso hijo.
- **InputStream**: flujo de entrada del proceso hijo. El stream está conectado por un pipe a la salida estándar (stdout) del proceso hijo.
- **ErrorStream**: flujo de error del proceso hijo. El stream está conectado por un pipe a la salida estándar (stderr) del proceso hijo.

Utilizando estos streams, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida.
Hay que tener en cuenta que en algunos sistemas operativos, el tamaño de los buffers de entrada y salida que corresponde a stdin y stdout está limitado.

Comunicación de procesos utilizando un *buffer*

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

public class ComunicationBetweenProcess {

    public static void main(String args[]) throws IOException {

            Process process = new ProcessBuilder(args).start();
        InputStream is = process.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line;

        System.out.println("Salida del proceso " +
                Arrays.toString(args) + ":");

        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }

    }
}
```

Además de la posibilidad de comunicarse mediante flujos de datos, existen otras alternativas para la comunicación de procesos:

- Usando sockets
- Utilizando JNI (Java Native Interface)
- Librerías de comunicación no estándares entre procesos en Java que permiten aumentar las capacidades del estándar Java.
    - Memoria compartida: se establece una región de memoria compartida entre varios procesos a la cual pueden acceder todos ellos.
    - Pipes: permite a un proceso hijo comunicarse con su padre a través de un canal de comunicación sencillo.
    - Semáforos: mecanismo de bloqueo de un proceso hasta que ocurra un evento.

1.7 SINCRONIZACIÓN DE PROCESOS

Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.
Concretamente, los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueante.

1.7.1 ESPERA DE PROCESOS (OPERACIÓN WAIT)

Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo y obtener su información de finalización mediante la **operación wait.**

- Dicha operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante exit.
- Como resultado se recibe la información de finalización del proceso hijo. Dicho valor de retorno se especifica mediante un número entero. El valor de retorno indica como resultó la ejecución (No son mensajes que se pasan padre e hijo). Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

Mediante **waitFor()** de la clase Process el padre espera bloqueado hasta que el hijo finalice su ejecución, volviendo inmediatamente si el hijo ha finalizado con anterioridad o si alguien le interrumpe (en este caso se lanza la interrupción InterruptedException). Además se puede utilizar exitValue() para obtener el valor de retorno que devolvió un proceso hijo.

Implementación de sincronización de procesos

```
import java.io.IOException;
import java.util.Arrays;

public class ProcessSincronization {

    public static void main(String[] args)
          throws IOException, InterruptedException {

    try{
          Process process = new ProcessBuilder(args).start();
          int retorno = process.waitFor();
          System.out.println("Comando " +  Arrays.toString(args)
               +  " devolvió: " + retorno);
    } catch (IOException e) {
          System.out.println("Error ocurrió ejecutando el
                              comando. Descripción: " + e.getMessage());
    } catch (InterruptedException e) {
          System.out.println("El comando fue interrumpido.
                              Descripción del error: " + e.getMessage());
    }
    }
}
```

1.8 PROGRAMACIÓN MULTIPROCESO

La programación concurrente permite que diferentes procesos se vayan alternando en la CPU eficientemente. La multiprogramación puede:

- Producirse entre procesos totalmente independientes (procesador de textos, navegador, reproductor de música, etc.)
- Producirse entre procesos que cooperan entre sí para realizar una tarea común.

El sistema operativo se encarga de proporcionar multiprogramación. Sin embargo, si se pretende realizar procesos que cooperen entre sí, debe ser el propio desarrollador quien lo implemente utilizando la comunicación y sincronización de proceso.

Para realizar un programa multiproceso cooperativo, se deben seguir las siguientes fases:

1. **Descomposición funcional**. Es necesario identificar previamente las diferentes funciones que debe realizar la aplicación y las relaciones existentes entre ellas.
2. **Partición.** Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos.
3. **Implementación.** Una vez realizada la descomposición y la partición se realiza la implementación. Java permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación

1.8.1 CLASE PROCESS

A la hora de realizar un algoritmo multiproceso en Java se utiliza la **clase Process.**

Método	Tipo de retorno	Descripción
getOutputStream()	OutputStream	Obtiene el flujo de salida del proceso hijo conectado al *stdin*
getInputStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al *stdout* del proceso hijo
getErrorStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al *stderr* del proceso hijo
destroy()	void	Implementa la operación *destroy*
waitFor()	int	Implementa la operación *wait*
exitValue()	int	Obtiene el valor de retorno del proceso hijo

1.9 EJERCICIO PRÁCTICO.

En este caso práctico se va a desarrollar una solución multiproceso al problema de sincronizar y comunicar dos procesos hijos creados a partir de un proceso padre. La idea es escribir una clase Java que ejecute dos comandos (cada hijo creado ejecutará uno de ellos) con sus respectivos argumentos y redireccione la salida estándar del primero a la entrada estándar del segundo. Por sencillez, los comandos y sus argumentos irán directamente escritos en el código del programa para no complicar demasiado el problema.

El siguiente ejemplo muestra la ejecución de los comandos ls –la y tr “d” “D” en Unix (el resultado debería ser el mismo que el de ejecutar en la shell de Linux o Mac ls –la | tr “d” “D”):

```
total 6
Drwxr-xr-x 5 user users 4096 2011-02-22 10:59 .
Drwxr-xr-x 8 user users 4096 2011-02-07 09:26 ..
-rw-r--r-- 1 user users 30 2011-02-22 10:59 a.txt
-rw-r--r-- 1 user users 27 2011-02-22 10:59 b.txt
Drwxr-xr-x 2 user users 4096 2011-01-24 17:49 Dir3
Drwxr-xr-x 2 user users 4096 2011-01-24 11:48 Dir4
```

1.10 EJERCICIO PROPUESTO.

Escribe una clase llamada Ejecuta que reciba como argumentos el comando y las opciones del comando que se quiere ejecutar. El programa debe crear un proceso hijo que ejecute el comando con las opciones correspondientes mostrando un mensaje de error en el caso de que no se realiza correctamente la ejecución. El padre debe esperar a que el hijo termine de informar si se produjo alguna anomalía en la ejecución del hijo.

1.11 EJERCICIO PROPUESTO 2.

Escribe un programa Aleatorios que haga lo siguiente:

- Cree un proceso hijo que está encargado de generar números aleatorios. Para su creación puede utilizarse cualquier lenguaje de programación generando el ejecutable correspondiente. Este proceso hijo escribirá en su salida estándar un número aleatorio del 0 al 10 cada vez que reciba una petición de ejecución por parte del padre. Nota: no es necesario utilizar JNI, solamente crear un ejecutable y llamar correctamente al mismo desde Java.
- El proceso padre lee líneas de la entrada estándar y por cada línea que lea solicitará al hijo que le envíe un número aleatorio, lo leerá y lo imprimirá en pantalla.
- Cuando el proceso padre reciba la palabra “fin”, finalizará la ejecución del hijo y procederá a finalizar su ejecución:

Ejemplo de ejecución:
ab (enter)
7
abcdef (enter)
1
Pepe (enter)
6
fin (enter)

1.12 EJERCICIO PROPUESTO 3.

Escribe una clase llamada Mayúsculas que haga lo siguiente:

- Cree un proceso hijo.
- El proceso padre y el proceso hijo se comunicarán de forma bidireccional utilizando streams.
- El proceso padre leerá líneas de su entrada estándar y las enviará a la entrada estándar del hijo (utilizando el OutputStream del hijo).
- El proceso hijo leerá el texto por su entrada estándar, lo transformará todo a letras mayúsculas y lo imprimirá por su salida estándar. Para realizar el programa hijo se puede utilizar cualquier lenguaje de programación generando un ejecutable.
- El padre imprimirá en pantalla lo que recibe del hijo a través del InputStream del mismo.

Ejemplo de ejecución:
hola (enter)
HOLA
mundo (enter)
MUNDO