Unidad Didáctica 15: Programación multiproceso

1. Conceptos Básicos

- **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.
- **Programa**: se puede considerar un programa a toda la información (tanto código como datos) almacenada en un disco de una aplicación que resuelve una necesidad concreta para los usuarios. Un programa se compone de un conjunto de instrucciones que se ejecutan siguiendo una secuencia.
- **Proceso:** el concepto central de cualquier sistema operativo es el proceso, una abstracción de un programa en ejecución.Cuando un programa es leído del disco por el núcleo y se carga en la memoria para su ejecución se convierte en un proceso.Podremos decir de manera simplificada que "proceso" es la instancia de un programa en ejecución por medio de una llamada al sistema operativo. Esto incluye tres cosas:
  1) *Un contador del programa*: lo que nos indica por dónde se está ejecutando
  2) *Una imagen de memoria:* es el espacio de memoria que el proceso está utilizando.
  3) *Estado del procesador:* se define como el valor de los registros del procesador sobre los cuales se está ejecutando.

  La clasificación más elemental de los procesos distingue entre *procesos de usuario* y *procesos de sistema*.

Un proceso de usuario se crea a partir de una ejecución directa o mediante una aplicación originada por el usuario.

Un *proceso de sistema* forma parte del sistema operativo, es decir, realizan operaciones de acceso a recursos de entrada/salida o de apoyo a la ejecución de otros procesos. Los procesos de sistema pueden ser:

- *Permanentes*: arrancan con el sistema y permanecen activos hasta que se apaga.
- *Transitorios*: la duración está determinada por la realización de una tarea en concreto, y finaliza una vez que la tarea se ha completado.

- **Ejecutable:** un fichero ejecutable contiene la información necesaria para crear un proceso a partir de los datos almacenados de un programa. Es decir, llamaremos "ejecutable" al fichero que permite poner el programa en ejecución como proceso.
- **Demonio:** Proceso no interactivo que está ejecutándose continuamente en segundo plano, es decir, es un proceso controlado por el sistema sin ninguna intermediación del usuario. Suelen proporcionar un servicio básico para el resto de procesos.

2. Estado de un proceso.

Como ya se ha dicho anteriormente, el sistema operativo (en adelante SO) es el encargado de poner en ejecución y gestionar los procesos. Para su correcto funcionamiento, a lo largo de su ciclo de vida, los procesos pueden cambiar de estado. Es decir, a medida que se ejecuta un proceso, dicho proceso pasará por varios estados, ya sea por su propia ejecución o por intervención del SO.

Estados en el ciclo de vida de un proceso.

1. **Nuevo.** Proceso nuevo, creado.
2. **Listo.** Proceso que está esperando a la CPU para ejecutar sus instrucciones (detenido de forma temporal para que se ejecute otro proceso)
3. **En ejecución.** Proceso que actualmente, está en turno de ejecución en la CPU.
4. **Bloqueado. No** se puede ejecutar debido a la ocurrencia de algún evento externo, por ejemplo está a la espera de que finalice una E/S.
5. **Suspendido.** Proceso que se ha llevado a la memoria virtual para liberar, un poco, la RAM del sistema.
6. **Terminado.** Proceso que ha finalizado y ya no necesitará más la CPU.

OBSERVACIÓN: Todo proceso en ejecución tiene que estar cargado en la RAM física del equipo o memoria principal, así como todos los datos que necesite.

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, es frecuente que los procesos deban interactuar con otros. Un proceso podría generar cierta salida que fuera utilizada por otro. Según las velocidades relativas de los dos procesos, podría ocurrir que el segundo estuviera listo para ser ejecutado, pero que no hubiera datos esperando para ser utilizados por él. Debe entonces bloquearse hasta

disponer de ciertos datos.

Cuando un proceso se bloquea, lo hace porque desde el punto de vista lógico no puede continuar, esto ocurre, por lo general, porque espera ciertos datos que no están disponibles todavía. También es posible que se detenga un proceso que esté listo y se pueda ejecutar, debido a que el sistema operativo ha decidido asignar la CPU a otro proceso.

Los procesos pueden tomar cualquiera de los estados mencionados y la transición entre uno y otro estado tiene su explicación.

Las siguientes son transiciones posibles:

- ***Ejecución - Listo***: Un proceso pasa de ejecución a listo, cuando se le acaba el tiempo asignado por el planificador de procesos del sistema, en este momento el sistema debe asignar el procesador a otro proceso.
- ***Listo - Ejecución***: Un proceso pasa de listo a ejecución cuando el sistema le otorga un tiempo de CPU.
- ***Ejecución - Bloqueado***: Un proceso pasa de ejecución a bloqueado cuando ejecuta una instrucción que implica la espera de un evento, por ejemplo debe esperar que ocurra efectivamente la E/S, espera por la activación de un semáforo, etc.
- ***Bloqueado - Listo***: Un proceso pasa de bloqueado a listo cuando el evento externo que esperaba sucede.

3. Planificación de procesos por el Sistema Operativo

Notar que hay procesos en el equipo cuya ejecución es crítica para el sistema, por lo que, no siempre pueden estar esperando a que les llegue su turno de ejecución, haciendo cola. Por ejemplo, el propio SO es un programa, y por lo tanto un proceso o un conjunto de procesos en ejecución. Se le da prioridad, evidentemente a los procesos del SO, frente a los procesos del usuario.

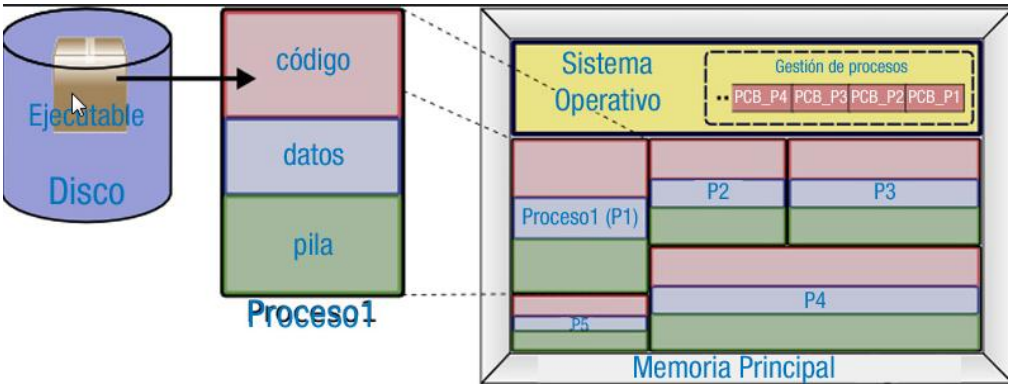
En esta gestión de procesos destacamos dos componentes del SO: el cargador y el planificador.

El *cargador* es el encargado de crear los procesos. Para cada proceso que se inicia el cargador, realiza las siguientes tareas:

a) Carga el proceso en memoria principal. Reserva un espacio en la RAM para el proceso. En ese espacio, copia las instrucciones del fichero ejecutable de la aplicación, las constantes, y deja un espacio para los datos (variables) y la pila (llamadas a funciones). Un proceso, durante su ejecución, no podrá hacer referencia a direcciones que se encuentren fuera de su espacio de memoria; si lo intenta, el SO lo detectará y generará una excepción (produciendo, por ejemplo, los típicos pantallazos azules de Windows).

b) Crea una estructura de información llamada PCB (Bloque de Control de Proceso). La información del PCB, es única para cada proceso y permite controlarlo. Esta información, también la utilizará el planificador. Entre otros datos, el PCB estará formado por:
   - Identificador del proceso o PID. Es un número único para cada proceso.
   - Estado actual del proceso: en ejecución, listo, bloqueado, finalizando.
   - Espacio de direcciones de memoria donde comienza la zona de memoria reservada al proceso y su tamaño.

- Información para la planificación: prioridad, quantum, estadísticas, ...
- Información para el cambio de contexto: valor de los registros de la CPU, entre ellos elcontador de programa y el puntero a pila. Esta información es necesaria para poder cambiar la ejecución de un proceso a otro.
- Recursos utilizados. Ficheros abiertos, conexiones, …



Una vez que el proceso ya está cargado en memoria, será el *planificador* el encargado de tomar las decisiones relacionadas con la ejecución de los procesos. Se encarga de decidir qué proceso se ejecuta y cuánto tiempo se ejecuta.

Los tipos básicos de estructuras de almacenamiento de un proceso son la pila y la cola.

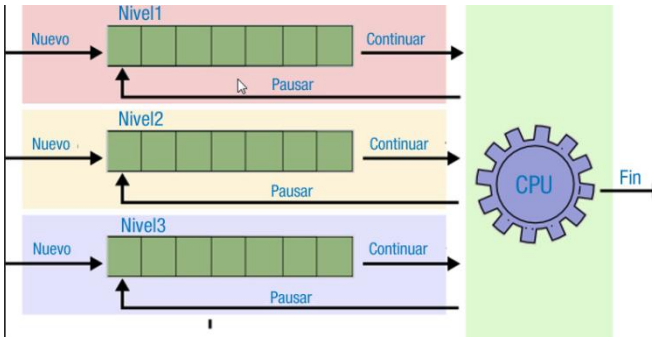
- Una pila es una estructura de datos en la que sus elementos se almacenan de forma que el último que entra es el primero que sale. Esta forma de almacenamiento se llama LIFO (Last In FirstOut) y se basa en dos operaciones: *push*, que introduce un elemento en la pila y *pop*, que desapila el último elemento introducido. En este sentido, una pila puede verse como un saco. El primer elemento que se introduce va al fondo del saco y, por tanto, es el último que sale.
- Una cola es una estructura de datos que sigue un almacenamiento de tipo FIFO (First In FirstOut), dónde el primer elemento que se almacena es el primero que sale. La operación *push* se realiza por un extremo de la cola, mientras *pop* se realiza por el otro. Esto se puede comparar con la típica cola de espera en la caja de un supermercado.

El planificador,que es parte del SO, toma las decisiones a partir de las formas anteriores de almacenamiento creando su algoritmo de planificación.

Los algoritmos más importantes son:

- **Round-Robin**. Este algoritmo de planificación favorece la ejecución de procesos interactivos. Es aquél en el que cada proceso puede ejecutar sus instrucciones en la CPU durante un quantum. Si no le ha dado tiempo a finalizar en ese quantum, se coloca al final de la cola de procesos listos, y espera a que vuelva su turno de procesamiento. Así, todos los procesos listos en el sistema van ejecutándose poco a poco.

- **Por prioridad**. En el caso de Round-Robin, todos los procesos son tratados por igual. Pero existen procesos importantes, que no deberían esperar a recibir su tiempo de procesamiento a que finalicen otros procesos de menor importancia. En este algoritmo, se asignan prioridades a los distintos procesos y la ejecución de estos, se hace de acuerdo a esa prioridad asignada. Por ejemplo: el propio planificador tiene mayor prioridad en ejecución que los procesos de usuario.
- **Múltiples colas**. Es una combinación de los dos anteriores y el implementado en los sistemas operativos actuales. Todos los procesos de una misma prioridad, estarán en la misma cola. Cada cola será gestionada con el algoritmo Round-Robin. Los procesos de colas de inferior prioridad no pueden ejecutarse hasta que no se hayan vaciado las colas de procesos de mayor prioridad.



En la planificación (scheduling) de procesos se busca conciliar los siguientes objetivos:

- Equidad. Todos los procesos deben poder ejecutarse.
- Eficacia. Mantener ocupada la CPU un 100 % del tiempo.
- Tiempo de respuesta. Minimizar el tiempo de respuesta al usuario.
- Tiempo de regreso. Minimizar el tiempo que deben esperar los usuarios de procesos por lotes para obtener sus resultados.
- Rendimiento. Maximizar el número de tareas procesadas por hora.

Al realizar el cambio de un proceso a otro, el planificador tiene que guardar el estado en el que se encuentra el microprocesador y cargar el estado en el que estaba el microprocesador cuando cortó la ejecución de otro proceso, para continuar con ese. Esto es lo que se conoce como *cambio de contexto*.

Esta rutina realiza las siguientes operaciones en el orden indicado:

1. Salvar el estado del proceso que se estaba ejecutando. El estado, también denominado contexto, consiste en los valores de todos los registros del microprocesador. Se copian en la memoria principal.
2. Seleccionar otro proceso para ejecutar. Entre todos los programas que estén preparados para ejecutarse, la rutina selecciona uno de ellos siguiendo algún algoritmo equitativo.
3. Restaurar el estado del proceso seleccionado. Para ello, se toma el estado asociado al proceso seleccionado previamente copiado en la memoria principal y se vuelca en los registros del microprocesador.
4. Ejecutar el proceso seleccionado. La rutina termina su ejecución saltando a la instrucción que estaba pendiente de ejecutar en el programa seleccionado.

4. Programación paralela, distribuida y concurrente

Con un único procesador, solamente puede ejecutarse una instrucción en un instante de tiempo.

La frase anterior parece una perogrullada, pero no lo es. Posiblemente estáis frente al ordenador leyendo este texto desde vuestro navegador web al mismo tiempo que escucháis música o se está descargando algo de internet. El ordenador está ejecutando varios programas simultáneamente, cuando en realidad sólo puede ejecutar una instrucción – según hemos afirmado al principio.

Esto sucede porque el ordenador tiene tal potencia de proceso que nos da la sensación de estar realizando ambas tareas al mismo tiempo, pero simplemente se trata de una ilusión: el ordenador va intercalando la ejecución con la velocidad adecuada para ofrecernos una percepción de simultaneidad. A la capacidad de ejecutar múltiples programas a la vez se le llama multitarea o multiproceso.

A medida que avanza la tecnología, los usuarios somos más exigentes, por lo que pedimos mayor capacidad al sistema operativo para que nos permita ejecutar múltiples programas a la vez, con la mayor velocidad posible y que sea lo más cómodo para nosotros. Claro está, todo esto va complicando la programación de estos sistemas operativos y, actualmente, ya es muy difícil encontrar sistemas que no incluyan la multitarea dentro de sus características.

En sistemas operativos que nos ofrecen multitarea/multiproceso podemos ejecutar varios programas (“ejecutar varios programas” quiere decir “tener varios procesos”) a la vez.

La programaciónactual permite la posibilidad de tener en ejecución al mismo tiempo múltiples tareas interactivas. Dichas tarea se pueden ejecutar:

- En un único procesador (**multiprogarmación**). En este caso, aunque para el usuario parezca que varios procesos se ejecutan al mismo tiempo, si solamente existe un único procesador, 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 periodo corto de tiempo (del orden de milisegundos). Esto permite que en un segundo se ejecuten múltiples procesos, creando en el usuario la precepción de que múltiples programas se están ejecutando al mismo tiempo. Este concepto se denomina **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 ejecutan al mismo tiempo.
- Varios núcleos en un mismo procesador **(multitarea**). La existencia de varios núcleos o *cores* en un ordenador es cada vez mayor, apareciendo en *Dual Cores, QuadCores,* en muchos casos i3,i5 e i7, etc. 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. En este caso todos los *cores* comparten la misma memoria por lo que es posible utilizarlos de forma coordinada mediante lo que se conoce por **programación paralela.**

- Varios **ordenadores distribuidos en red**. Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria. La gestión de los mismos forma parte de lo que se denomina **programación distribuida.** La programación distribuida posibilita la utilización de un gran número de dispositivos (ordenadores) de forma paralela, lo que permite alcanzar elevadas mejoras en el rendimiento de la ejecución de programas distribuidos. Sin embargo, como cada ordenador posee su propia memoria, imposibilita que los procesos puedan comunicarse compartiendo memoria, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red que los interconecte.

5. Creación y finalización de procesos

El sistema operativo es el encargado de crear los nuevos procesos siguiendo las directrices del usuario. Así, 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, ya que es el único que puede acceder a los recursos del ordenador, el nuevo proceso se crea siempre por petición de otro proceso. La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario.

En este sentido, cualquier 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. Cuando se arranca el ordenador, y se carga en memoria el *kernel* del sistema a partir de su imagen en disco, se crea el resto de procesos de forma jerárquica, estableciendo padres, hijos, abuelos…

Para identificar a los procesos, los sistemas operativos suelen utilizar un identificador de procesos (PID) unívoco para cada procesos.

Siguiendo el vínculo entre procesos establecido en 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. A la operación de creación de un nuevo proceso la denominaremos *create.*

Cuando se crea un nuevo proceso tenemos que saber que padre e hijo se ejecutan concurrentemente. Ambos procesos comparten la CPU y se irán intercambiando siguiendo la política de planificación del sistema operativo para proporcionar multiprogramación. Si el proceso padre tiene que esperar hasta que el hijo termine su ejecución para poder continuar la suya con los resultados obtenidos por el hijo, puede hacerlo con la operación *wait.*

Como se ha visto al inicio del capítulo, los procesos son independientes y tienen su propio espacio de memoria asignado, llamado imagen de memoria. Padres e hijos son procesos y, aunque tengan un vínculo especial, mantienen esta restricción, ambos usan espacios de memoria independientes. En general, parece que el hijo ejecuta un programa diferente al padre, pero en algunos sistemas operativos esto no tiene por qué ser así. Por ejemplo, mientras que en sistemas tipo Windows existe una función *createProcess()* que crea un nuevo proceso a partir de un programa distinto al que está en ejecución, en sistemas tipo UNIX, la operación a utilizar es *fork(),* que crea un proceso hijo con un duplicado del espacio de direcciones del padre, es decir, un duplicado del programa que se ejecuta desde la misma posición. Sin embargo, en ambos casos, los padres e hijos(aunque sean un duplicado en el momento de la creación de sistemas tipos UNIX) son independientes y las modificaciones que uno haga en su espacio de memoria, como escritura de variables, no afectarán al otro.

Como padre e hijo tienen espacios de memoria independientes, pueden compartir recursos para intercambiarse información. Estos recursos pueden ir desde ficheros abiertos hasta zonas de memoria compartida. La memoria compartida es una región de memoria a la que pueden acceder varios procesos cooperativos para compartir información. Los procesos se comunican escribiendo y leyendo datos en dicha región. El sistema operativo solamente interviene a la hora de crear y establecer los permisos de los procesos que pueden acceder a dicha zona. Los procesos son los responsables del formato de los datos compartidos y de su ubicación.

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. En general, es el propio procesos el que le indica al sistema operativo mediante una operación denominada *exit* que quiere terminar, pudiendo aprovechar para mandar información respecto a su finalización al proceso padre en ese momento.

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. Entre estos motivos podría darse que el hijo excediera el uso de algunos recursos o que la funcionalidad asignada al hijo ya no sea necesaria por algún motivo.

Para ello puede utilizar la operación *destroy*. Esta relación de dependencia entre padre e hijo, lleva a casos como que si el padre termina, en algunos sistemas operativos no se permita que sus hijos continúen la ejecución, produciéndose lo que se denomina "terminación en cascada".

En definitiva, cada sistema operativo tienen unas características únicas, y la gestión de los procesos es diferente en cada uno de ellos. Por simplificación y portabilidad, evitando así depender del sistema operativo sobre el cual se esté ejecutando hemos decidido explicar la gestión de procesos para la máquina virtual de Java (JVM).JVM es un entorno de ejecución ligero y gratuito multiplataforma que permite la ejecución de binarios o *bytecode* del lenguaje de programación Java sobre cualquier sistema operativo, salvando las diferencias entre ellos. La idea que hay detrás de ello se conoce como *Write Once, RunAnywhere.*

5.1. Creación de procesos.

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 subyacente donde se está ejecutando la JVM y devuelven un obeto Java de la clase *Process* que puede ser utilizado para controlar dicho proceso.

- ***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().**
- ***ProcessRuntime*.exec (String[] cdmarray, String[] envp, File dir):** ejecuta el comando especificado y argumentos en cdmarray en un proceso hijo independiente con el entorno envp y el directorio de trabajo especificado en dir.

Ambos métodos comprueban que el comando a ejecutar es un comando o ejecutable válido en el sistema operativo subyacente sobre el que ejecuta la JVM. El ejecutable se ha podido obtener mediante la compilación de código en cualquier lenguaje de programación. Al final, crear un nuevo proceso depende del sistema operativo en concreto que esté ejecutado por debajo de la JVM. En este sentido, pueden ocurrir múltiples problemas, como:

-No encuentra el ejecutable debido a la ruta indicada.

-No tiene permisos de ejecución.

-No ser un ejecutable válido en sistema.

En la mayoría de los casos, se lanza una excepción dependiente del sistema en concreto, pero siempre será una subclase de *IOException.*

```
//creación de proceos con la clase ProcessBuilder
package procesos_process_builder;

public class procesos_03_process_builder {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		String comando="calc";

		ProcessBuilder pb= new ProcessBuilder(comando);

		try {
			pb.start();
		} catch (Exception e){
			System.out.println("Error en: " + comando);
			e.printStackTrace();
		}

	}

}
```

Ejemplo 1: creación de un proceso usando ProcessBuilder

```
//Runtime representa el entorno de ejecución de la aplicación
//Runtime getRuntime() devuelve el objeto runtime asociado con la aplicación java en curso

//Process exec(comando) ejecuta la orden especificada en "comando"
package procesos_01;

public class procesos_01 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Runtime r=Runtime.getRuntime();
		String comando="calc";

		Process p;

		try {
			p=r.exec(comando);
		} catch (Exception e){
			System.out.println("Error en: " + comando);
			e.printStackTrace();
		}

	}

}
```

Ejemplo 2: creación de un proceso mediante Runtime

5.2. 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. En caso de Java, los recursos correspondientes los eliminará el *garbagecollector* cuando considere.

Si no se fuerza la finalización de la ejecución del proceso hijo de forma anómala, el proceso hijo realizará su ejecución completa terminado y liberando sus recursos al finalizar. Eso se produce cuando el hijo realiza la operación *exit* para finalizar su ejecución.

```
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);
              }
    }
```

Ejemplo 3: creación de un proceso mediante Runtime para después destruirlo

5.3. Planificación de procesos.

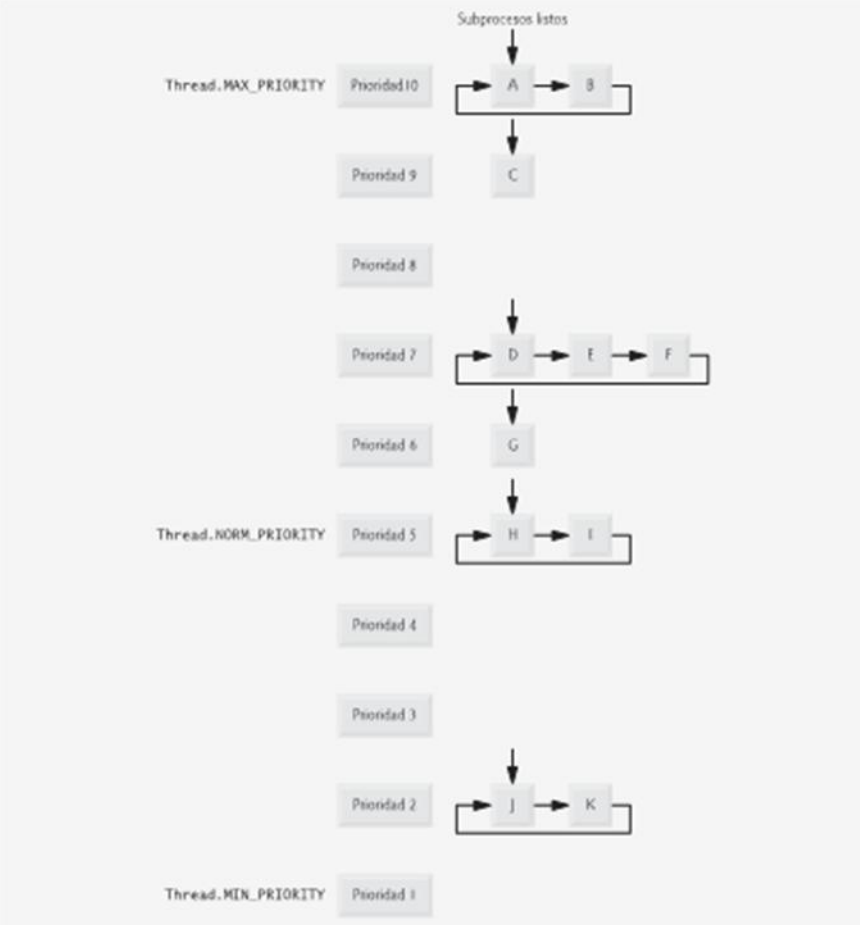
Todo proceso en Java tiene una prioridad, la cual ayuda al sistema operativo a determinar el orden en el que se programan los procesos. Las prioridades de Java varían entre MIN_PRIORITY (una constante de 1) y MAX_PRIORITY (una constante de 10). De manera predeterminada cada proceso recibe la prioridad NORM_PRIORITY (una constante de 5). Cada nuevo subproceso hereda la prioridad del proceso que lo creó. De manera informal, los procesos de mayor prioridad son más importantes para un programa, y se les debe asignar tiempo del procesador antes que a los procesos de menor prioridad. Sin embargo, las prioridades de los procesos no garantizan el orden en que se ejecutan.

La mayoría de los sistemas soportan los intervalos de tiempo, que permiten a los procesos de igual prioridad compartir un procesador. Sin el intervalo de tiempo, cada proceso en un conjunto de procesos de igual

prioridad se ejecuta hasta completarse antes que otros procesos de igual prioridad tengan oportunidad de ejecutarse. Con el intervalo de tiempo, aun si un proceso no ha terminado de ejecutarse cuando expira su quantum, el procesador se quita del proceso y pasa al siguiente proceso de igual prioridad, si hay uno disponible.

El programador de procesos de un SO determina qué proceso se debe ejecutar a continuación. Una implementación simple del programador de procesos mantiene el proceso de mayor prioridad en ejecución en todo momento y, si hay más de un proceso de mayor prioridad, asegura que todos esos procesos se ejecuten durante un quantum cada uno, en forma cíclica.

La siguiente imagen:



Muestra una cola de prioridades multinivel para los procesos. Supongamos que hay un solo ordenador con un procesador, los procesos A y B se ejecutan cada uno durante un quantum, en forma cíclica hasta que los dos terminan su ejecución. Después el procesador dedica el resto del tiempo a los procesos restantes (a menos que esté listo otro proceso de prioridad 10). Entonces se ejecuta el proceso C hasta que termina (suponiendo que no entran procesos con mayor prioridad). Después los procesos los D, E y F se ejecutan cada uno durante un quantum, en forma cíclica hasta que todos terminan (si no entra otro proceso de mayor prioridad). Y continua así hasta que todos los procesos terminan de ejecutarse.

Cuando un proceso de mayor prioridad entra, el sistema operativo generalmente sustituye el proceso actual en ejecución para dar preferencia al proceso de mayor prioridad.

6. Comunicación entre procesos.

Para comunicar procesos, el intercambio de mensajes se puede hacer de dos maneras:

- Usando buffer de memoria (tuberías o *pipes*): el S.O. redirecciona la salida y entrada estándar de los procesos.
- Usando socket (más usado para intercambiar información entre distintas máquinas de una red): se crea un canal de comunicación directo entre dos procesos.

Por ahora nos vamos a centrar en la comunicación mediante flujos de datos o pipes, ya que es la más usual si trabajamos en la misma máquina.

Es importante recordar que un proceso es un programa en ejecución y, como cualquier programa, recibe información, la transforma y produce resultados. Por ello se usa la entrada estándar (stdin), la salida estándar (stdout) y la salida de error (stderr).La utilización de System.out y System.err en Java se puede ver como un ejemplo de utilización de estas salidas.

En la mayoría de los sistemas operativos, estas entradas y salidas en proceso hijo son una copia de las mismas entradas y salida que tendría el padre. De tal forma que si se llama a la operación *create*dentro de un proceso que lee de un fichero y muestra la salida estándar por pantalla, su hijo correspondiente leerá el fichero y escribirá en pantalla. En java, en cambio, el proceso hijo creado de la clase Process no tiene su propia interfaz de comunicación, por lo que el usuario o puede comunicarse con él directamente. Todas sus salidas y entradas de información (stdin,stdout y stderr) se redirigen al proceso padre a través de los siguientes flujos de datos o streams:

- 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 jioj. 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 de proceso hijo. Sin embargo, hay que saber que por defecto, para la JVM, stderr está conectado al mismo sitio que stdout. Si se desea tenerlos separados, lo que permite identificar errores de forma más sencilla, se puede utilizar el método *redirectErrorStream (boolea*n) de la clase *ProcessBuilde*r. Si se pasa un valor true como parámetro, los flujos de datos correspondientes a stderr y stdout en la JVM serán diferentes y representarán la salida estándar y la salida de error del proceso de forma correspondiente.

Utilizando estos *streams*, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida que este genere comprobando los errores.

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. En este sentido, un fallo al leer o escribir en los flujos de

entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee. Por eso, en Java se suele realizar la comunicación padre-hijo a través de un buffer utilizando los streams vistos.

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.

OBS: si el padre pide leer de la salida del hijo *stdout* a través de su *InputStream*se bloquea hasta que el hijo le devuelve los datos requeridos. En este sentido, padre e hijo pueden sincronizarse de la forma adecuada.

6.1. Espera de procesos

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 a proceso padre hasta que el jijo 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 significa cómo resulto la ejecución. No tiene nada que ver con los mensajes que se pasan padre e hijo a través de los *streams.* Por conveniencia 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. El proceso hijo debe haber finalizado, si no, se lanza la excepción *IllegalThreadSteateException.*

```
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:" + e.getMessage());
        } catch (InterruptedException e) {
                System.out.println("El comando fue interrumpido. Descripción del error: " +
                e.getMessage());
        }
    }
}
```

Ejemplo : sincronización de procesos