Tema 1: Programación Multiprocesos - Resumen

Álvaro Díaz Barrios

03/11/2022

Contents

1 Conceptos básicos **2**

2 Programación concurrente **2**

3 Funcionamiento de un sistema operativo **2**

4 Procesos **2**
4.1 Estados de un proceso . . . 2
4.2 Colas de procesos . . . 3
4.3 Planificación de procesos . . . 3
4.4 Cambio de contexto . . . 3

5 Gestión de procesos **3**
5.1 Árbol de procesos . . . 3
5.2 Operaciones básicas con procesos . . . 4
5.2.1 Clases Runtime y Process . . . 4
5.2.2 Clase ProcessBuilder . . . 4

6 Condiciones de Bernstein **5**
6.1 Tipos de dependencias . . . 5

7 Referencias rápidas **5**
7.1 Clase Runtime . . . 5
7.2 Clase Process . . . 6
7.3 Clase ProcessBuilder . . . 6

1 Conceptos básicos

- **Programa**: es toda la información (tanto código como datos) de una aplicación que resuelve una necesidad concreta para os usuarios.
- **Proceso**: es un programa en ejecución. Incluye:
  - **Un contador de programa**: indica por donde se está ejecutando el programa.
  - **Una imagen de memoria**: es el espacio de memoria que el proceso utiliza.
  - **Estado del procesador**: es el valor de los registros del procesador sobre los cuales se está ejecutando.
- **Ejecutable**: contiene la información necesaria para crear un proceso a parti de los datos almacenados de un programa.
- **Sistema operativo**: programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador.
- **Demonio**: es un proceso controlado por el sistema sin ninguna intermediación del usuario.

2 Programación concurrente

La computación concurrente permite la ejecución al mismo tiempo de múltiples tareas interactivas. Estas se pueden ejecutar en:

- **Un único procesador (multi-programación)**.
- **Varios núcleos en un mismo procesador (multi-tarea)**: cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo. El sistema operativo se encarga de gestionar los trabajos. Como todos los procesos comparten la misma memoria es posible la **programación paralela**.
- **Varios ordenadores distribuidos en red (programación distribuida)**: posibilita la utilización de un gran número de dispositivos de forma paralela, lo que permite alcanzar grandes mejoras en el rendimiento de la ejecución. Sin embargo imposibilita que los procesos puedan comunicarse compartiendo memoria.

3 Funcionamiento de un sistema operativo

Se denomina **kernel** a la parte del sistema operativo que realiza las funcionalidades más básicas. Al resto de programas se le denomina **programas del sistema**. El **kernel** es el responsable de gestionar los recursos del ordenador, permitiendo su uso a través de llamadas al sistema.

El **kernel** funciona a base de **interrupciones**, esta es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción. Cuando esta rutina se ha finalizado, se reanuda la ejecución del proceso en el mismo lugar donde se quedó cuando se interrumpió.

Las **llamadas al sistema** son **interfaces** que proporcionan el **kernel** para que los programas de usuario puedan hacer uso de forma segura determinadas partes del sistema.

El **modo dual** es una característica del hardware que permite al sistema operativo protegerse. Como su nombre indica, hay dos modos:

- **Modo usuario (1)**: utilizado para la ejecución de programas de usuario.
- **Modo kernel (0)**: se pueden ejecutar las instrucciones del procesador más delicadas.

4 Procesos

4.1 Estados de un proceso

- **Nuevo**: el proceso está siendo creado a partir del fichero ejecutable.
- **Listo**: el proceso no se encuentra en ejecución aunque está preparado para hacerlo.

- **En ejecución**: el proceso se está ejecutando.
- **Bloqueado**: el proceso está esperando que ocurra algún suceso. Cuando el suceso se dispara, el proceso pasa a **listo**, ya que tiene que ser planificado de nuevo por el sistema.
- **Terminado**: El proceso ha finalizado su ejecución y libera su imagen de memoria.

4.2 Colas de procesos

El sistema operativo organiza los procesos en varias colas, moviendo los mismos de unas a otras. Las cosas que encontramos son las siguientes:

- **Cola de procesos**: contiene todos los procesos del sistema.
- **Cola de procesos preparados**: contiene todos los procesos listos.
- **Colas de dispositivo**: que contiene los procesos que están a la espera de alguna operación E/S.

4.3 Planificación de procesos

El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas. Hay de dos tipos:

- **A corto plazo**: selecciona qué procesos de la **Cola de procesos preparados** pasará a ejecución. Debe tener algoritmos sencillos debido a la alta frecuencia con la que se llama.
  - **Planificación sin desalojo o cooperativo**: únicamente se cambia el proceso en ejecución si este se bloquea o termina.
  - **Planificación apropiativa**: cambia el proceso en ejecución si en cualquier momento en que un proceso se está ejecutando, otro proceso con mayor prioridad se puede ejecutar.
  - **Tiempo compartido**: cada cierto tiempo se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse.
- **A largo plazo**: selecciona qué procesos nuevos deben pasar a la cola de procesos preparados.

4.4 Cambio de contexto

Cuando se cambian de proceso en ejecución, el sistema operativo debe guardar el contexto en el que se encontraba el anterior proceso ejecutado para poder reanudarlo más tarde. Se conoce como **contexto**:

- 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.

5 Gestión de procesos

5.1 Árbol de procesos

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. Cualquier proceso en ejecución siempre depende del proceso que lo creó. A su vez, este nuevo proceso puede crear otros procesos que dependen de él, llegándose a formar así lo que llamamos **árbol de procesos**.

El **bloque de control de proceso (BCP)** es donde se almacena la información de un proceso, entre la que encontramos:

- Identificación del proceso **(PID)**.
- Estado del proceso.

- Contador de programa.
- Registros de la CPU.
- Información de planificación de CPU como la prioridad del proceso.
- Información de gestión de memoria.
- Información contable como la calidad de tiempo de CPU y tiempo real consumido.
- Información de E/S como la lista de dispositivos asignados, archivos abiertos, etc.

5.2 Operaciones básicas con procesos

Para operar con procesos en Java se usan 3 clases **Runtime**, **Process** y **ProcessBuilder**. Tanto **Runtime** como **ProcessBuilder** necesitaran del uso de **Process**.

5.2.1 Clases Runtime y Process

La **clase Runtime** la usaremos solamente para hacer uso de dos métodos: **getRuntime()** y **exec(String comando)**. El método **getRuntime()** nos devuelve un objeto de tipo Runtime que está asociado a la aplicación actual de java. Por otro lado, el método **exec(String comando)** recibe una cadena que representa el comando que queremos ejecutar y devuelve un objeto Process del proceso hijo.

De la **clase Process** haremos uso de los siguientes métodos:

- **getInputStream()**: devuelve un objeto InputStream conectado con la **salida** normal del sub-proceso.
- **getOutputStream()**: devuelve un objeto OutputStream conectado con la **entrada** normal del sub-proceso.
- **waitFor()**: obliga al hilo actual a esperar, hasta que el proceso representado por este objeto Process finalice. También devuelve un entero que es el valor de salida del sub-proceso.
- **isAlive()**: comprueba si el sub-proceso representado por el objeto Process está o no en ejecución. Devuelve un booleano.
- **getErrorStream()**: devuelve un objeto InputStream conectado con la **salida** de error del sub-proceso.
- **destroy()**: destruye el sub-proceso representado por el objeto Process.
- **exitValue()**: devuelve el valor de salida del el sub-proceso representado por el objeto Process.

5.2.2 Clase ProcessBuilder

La **clase ProcessBuilder** se usa para crear procesos sin la necesidad de la clase Runtime. Los métodos que usaremos de esta clase son los siguientes:

- **command()**: devuelve una List¡String¿ que contiene el programa del sistema operativo y argumentos.
- **command(String... comando)**: establece el programa del sistema operativo y los argumentos de este ProcessBuilder.
- **directory()**: devuelve un objeto File que represente el directorio de trabajo de este ProcessBuilder.
- **directory(File directorio)**: establece el directorio de trabajo de este ProcessBuilder.
- **environment()**: devuelve un objeto Map con las variables de entorno de este Process Builder.
- **redirectError(File archivo)**: redirecciona la salida estándar de error de este ProcessBuilder al archivo destino.
- **redirectError(ProcessBuilder.Redirect destino)**: redirecciona la salida estándar de error de este ProcessBuilder al destino.
- **redirectErrorStream(boolean redireccionarErrorStream)**: establece la propiedad redirectErrorStream de este ProcessBuilder.

- **redirectInput(File archivo)**: redirecciona la entrada estándar de este ProcessBuilder al archivo destino.
- **redirectInput(ProcessBuilder.Redirect origen)**: redirecciona la entrada estándar de este ProcessBuilder al destino.
- **redirectOutput(File archivo)**: redirecciona la salida estándar de este ProcessBuilder al archivo destino.
- **redirectOutput(ProcessBuilder.Redirect origen)**: redirecciona la salida estándar de este ProcessBuilder al destino.
- **start()**: comienza un nuevo proceso usando los atributos de este ProcessBuilder. Devuelve un objeto Process asociado al nuevo proceso.

6 Condiciones de Bernstein

Para poder determinar si dos conjuntos de instrucciones se pueden ejecutar concurrentemente, sea S_k un conjunto de instrucciones a ejecutar, llamamos:

- $L(S_k) = \{a_1, a_2, ...a_n\}$ al **conjunto de lectura**, formado por todas las variables cuyos valores son referenciados (se leen) durante la ejecución de las instrucciones S_k.
- $E(S_k) = \{b_1, b_2, ...b_m\}$ al **conjunto de escritura**, formado por todas las variables cuyos valores son actualizados durante la ejecución.

Para que se puedan ejecutar concurrentemente dos conjuntos de instrucciones S_i y S_j se debe cumplir que ninguno escriba lo que el otro lee o escribe:

1. $L(S_i) \cap E(S_j) = \emptyset$
2. $E(S_i) \cap L(S_j) = \emptyset$
3. $E(S_i) \cap E(S_j) = \emptyset$

6.1 Tipos de dependencias

1. **De datos**: cuando dos instrucciones hacen uso del mismo dato el orden de ejecución en son importante. Existen los siguientes tipos:
   (a) Dependencias de flujo.
   (b) Antidependencias.
   (c) Dependencias de salida.
   (d) Dependencias de E/S (fichero)
   (e) Desconocidas.
2. **Dependencias de control**: cuando el orden de ejecución no puede ser determinado estáticamente, sino sólo en tiempo de ejecución.
3. **Dependencias de recursos**: cuando recursos compartidos del procesador son demandados por instrucciones diferentes al mismo tiempo.

7 Referencias rápidas

7.1 Clase Runtime

Modificador y tipo	Método	Descripción
static Runtime	getRuntime()	devuelve un Runtime asociado a la aplicación actual de java
Process	exec(String comando)	ejecuta un comando

7.2 Clase Process

Modificador y tipo	Método	Descripción
abstract void	destroy()	mata el sub-proceso
abstract int	exitValue()	devuelve el valor de salida del sub-proceso
abstract InputStream	getErrorStream()	devuelve el inputStream conectado al error outputStream del sub-proceso
abstract InputStream	getInputStream()	devuelve el inputStream conectado al outputStream del sub-proceso
abstract OutputStream	getOutputStream()	devuelve el outputStream conectado al inputStream del sub-proceso
boolean	isAlive()	comprueba si el sub-proceso está ejecutándose
abstract int	waitFor()	hace que el hilo actual espere a que acabe el sub-proceso

7.3 Clase ProcessBuilder

Modificador y tipo	Método	Descripción
List< *String* >	command()	devuelve el programa del sistema operativo y argumentos.
ProcessBuilder	command(String... command)	establece el programa del sistema operativo y argumentos.
File	directory()	devuelve el directorio de trabajo de este ProcessBuilder.
ProcessBuilder	directory(File directorio)	establece el directorio de trabajo de este ProcessBuilder.
Map< *String*, *String* >	environment()	devuelve las variables de entorno
ProcessBuilder	redirectError(File archivo)	redirecciona la salida estándar de error
ProcessBuilder	redirectErrorStream(boolean b)	establece la propiedad redirectErrorStream
ProcessBuilder	redirectInput(File archivo)	redirecciona la entrada estándar
ProcessBuilder	redirectOutput(File archivo)	redirecciona la salida estándar de error
Process	start()	ejecuta el proceso