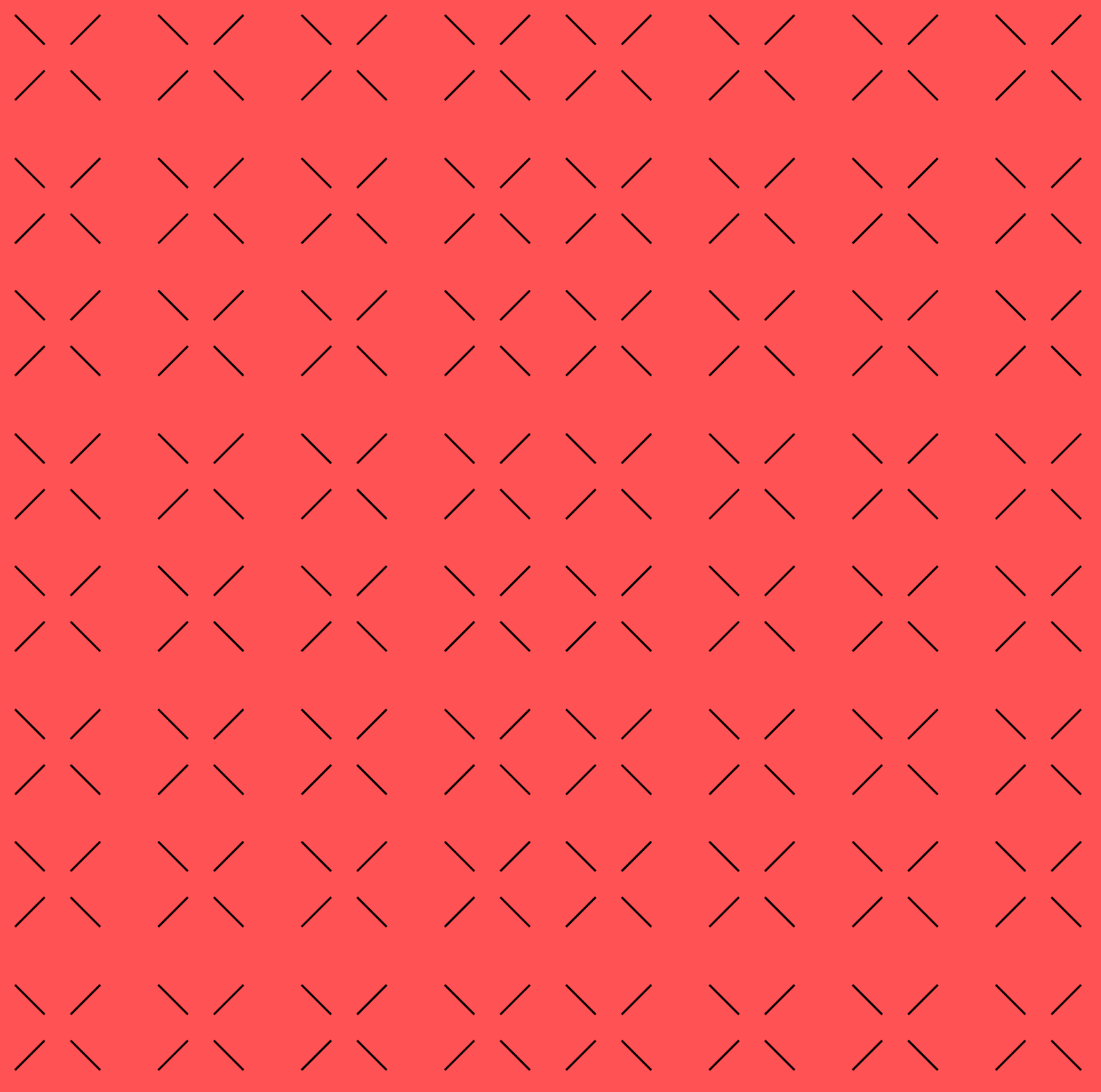


Unidad 1.1

Elementos de la programación concurrente



Índice

Sumario

1. DEFINICIONES.....	4
1.1. PROGRAMA.....	4
1.2. PROCESOS.....	4
1.3. SERVICIOS.....	5
1.4. HILOS.....	5
1.5. MULTIPROCESO.....	5
1.6. MULTIHILO.....	5
2. CONCURRENCIA Y DISTRIBUCIÓN.....	6
2.1. PROGRAMACIÓN CONCURRENTE Y PROGRAMACIÓN PARALELA.....	6
2.2. PROGRAMACIÓN PARALELA Y DISTRIBUIDA.....	7
2.3. HILOS.....	9
3. ALGORITMOS DE PLANIFICACIÓN.....	10
3.1. FCFS: First Come First Served.....	10
3.2. SJF: shortes Job First.....	12
3.3. SRT: Shortest Remaining Time.....	12
3.4. RR: Round Robin.....	13
3.5. Varias colas con realimentación.....	15

Licencia



Reconocimiento – NoComercial – CompartirIgual (by-nc-sa):

No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

1. DEFINICIONES

1.1. PROGRAMA

Conjunto de instrucciones que una computadora interpreta para resolver una tarea o problema. Las instrucciones tienen asociado un espacio de memoria para hacer el tratamiento de datos.

Actualmente a los programas también se les puede llamar aplicaciones o apps.

Definición de APP: es un tipo de software de computadora diseñado para realizar un grupo de funciones, tareas o actividades coordinadas para el beneficio del usuario. Como ejemplos de APP's tenemos: un procesador de textos, una hoja de cálculo, una aplicación de contabilidad, un navegador web, un reproductor multimedia, un simulador de vuelo aeronáutico o un editor de fotografías.

1.2. PROCESOS

Es un archivo que está en ejecución y bajo el control del sistema operativo. Es por ello la unidad de actividad básica que trata el Sistema Operativo, que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados.

Un proceso puede atravesar diversas etapas en su «ciclo de vida». Los estados en los que puede estar son:

- **En ejecución:** está dentro del microprocesador.
- **Pausado/detenido/en espera:** el proceso tiene que seguir en ejecución pero en ese momento el S.O. tomó la decisión de dejar paso a otro.
- **Interrumpido:** el proceso tiene que seguir en ejecución pero el usuario ha decidido interrumpir la ejecución.
- Existen otros estados pero ya son muy dependientes del sistema operativo concreto.

En resumen, los procesos son gestionados por el sistema operativo y están formados por:

- **Las instrucciones de un programa** destinadas a ser ejecutadas por el microprocesador.
- **Su estado** de ejecución en un momento dado, esto es, los valores de los registros de la unidad central de procesamiento para dicho programa.
- **Su memoria de trabajo** (memoria crítica), es decir, la memoria que ha reservado y sus contenidos.
- Otra información que permite al sistema operativo su planificación.

1.3. SERVICIOS

Un servicio es un proceso que no muestra ninguna ventana ni gráfico en pantalla porque no está pensado para que el usuario lo maneje directamente.

Habitualmente, un servicio es un programa que atiende a otro programa.

1.4. HILOS

Un hilo es un concepto más avanzado que un proceso: al hablar de procesos cada uno tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta.

Un proceso no tiene acceso a los datos de otros procesos. Sin embargo, un hilo sí accede a los datos de otro hilo. Esto complicaría algunas cuestiones a la hora de programar.

1.5. MULTIPROCESO

El multiproceso consiste en la ejecución de varios procesos diferentes de forma simultánea para la realización de una o varias tareas relacionadas o no entre sí. En este caso, cada uno de estos procesos es una aplicación independiente. El caso más conocido es aquel en el que nos referimos al Sistema Operativo (Windows, Linux, MacOS, . . .) y decimos que es multitarea puesto que es capaz de ejecutar varias tareas o procesos (o programas) al mismo tiempo.

1.6. MULTIHILLO

Hablamos de multihilo cuando se ejecutan varias tareas relacionadas o no entre sí dentro de una misma aplicación. En este caso no son procesos diferentes sino que dichas tareas se ejecutan dentro del mismo proceso del Sistema Operativo. A cada una de estas tareas se le conoce como hilo o thread (en algunos contextos también como procesos ligeros).

En ambos casos estaríamos hablando de lo que se conoce como Programación Concurrente. Hay que tener en cuenta que en ninguno de los dos casos la ejecución es realmente simultánea, ya que el Sistema Operativo es quién hace que parezca así, pero los ejecuta siguiendo lo que se conoce como algoritmos de planificación

2. CONCURRENCIA Y DISTRIBUCIÓN

2.1. PROGRAMACIÓN CONCURRENTE Y PROGRAMACIÓN PARALELA


Para entender la programación concurrente tenemos primero que distinguir entre:


- **Sistema monoprogramación:** Ejecuta el código de un único programa hasta que este haya concluido.
- **Programación concurrente:** puede soportar dos o más acciones en progreso (coexisten en memoria y se ejecutan a turnos)
- **Programación paralela:** Soporta dos o más acciones ejecutándose simultáneamente.

Para que un programa sea paralelo, no solo debe ser concurrente, sino que también debe estar diseñado para correr en un medio con hardware paralelo (GPU's, procesadores multi-core, etc).

Puede ser visto como que la concurrencia es la propiedad de un programa, mientras que el paralelismo es la forma en la que se ejecuta un programa concurrente.

Términos importantes:

 **Quantum:** Tiempo límite que el sistema operativo asigna a la ejecución de un proceso. Cuando se acaba el quantum se pasa al siguiente proceso.

 **Cambio de contexto:** Lo lleva a cabo el S.O. para salvaguardar los datos actuales de ejecución del proceso en curso y restaurar los del siguiente proceso a ejecutar

En la programación concurrente hay pues dos o mas procesos que se está ejecutando a la vez, pero al tener sólo un procesador deben ejecutarse por turnos. El quantum es el tiempo que tiene para ejecutarse, cuando se acaba pasa el siguiente proceso.

Hay que tener en cuenta que el quantum en si no puede ser ni demasiado grande, ni demasiado pequeño.

- **Problema si un quantum es demasiado grande:** Podría acabar en un sistema de monoprogramación, cuando acaba el proceso1 se empieza a ejecutar el proceso2. El quantum es tan grande que le da tiempo a acabar al proceso1.
- **Problema si un quantum es demasiado pequeño:** Cada vez que a un proceso se le acaba el tiempo de su quantum, "se pierde tiempo" en el cambio de contexto, para que el siguiente proceso entre a ejecutarse. Si el quantum es muy pequeño hay muchas más cambios de contexto y si miramos el tiempo final (ejecución de los procesos + el tiempo "perdido" en los cambios de proceso), sería más grande que el tiempo que hubiéramos tardado en ejecutar los procesos uno tras otro.

En la programación paralela los procesos se dividen en varias líneas de ejecución que son tratadas por diversos procesadores. Dicha división se puede hacer de forma:

- **Manual:** Se programan las partes que se quieren ejecutar de forma paralela. Para ello se usa lo que se denomina por hilos o threads.
- **Automática:** Está en vías de desarrollo. Se trata que en vez de una persona que decide qué partes del programa se pueden hacer de forma paralela, dejaríamos esa tarea al compilador que crearía los diferentes hilos.

Ejemplo de concurrencia:

Imagina una aplicación de descarga de música, en la cual puedes descargar un número determinado de canciones al mismo tiempo, cada canción es independiente de la otra, por lo que la velocidad y el tiempo que tarda en descargarse cada una no afectará al resto de canciones. Esto lo podemos ver como un proceso concurrente, ya que cada descarga es un proceso totalmente independiente del resto.

Ejemplo de paralelismo:

Imagina la clásica página de viajes, donde nos ayudan a buscar el vuelo más barato o las mejores promociones, para hacer esto, la página debe de buscar al momento en cada aerolínea el vuelo más barato, con menos conexiones, etc. Para esto puedo hacerlo de dos formas, buscar secuencialmente en cada aerolínea las mejores promociones (mucho tiempo) o utilizar el paralelismo para buscar al mismo tiempo las mejores promociones en todas las aerolíneas.

2.2. PROGRAMACIÓN PARALELA Y DISTRIBUIDA

Dentro de la programación concurrente tenemos la paralela y la distribuida:

- En general se denomina «programación paralela» a la creación de software que se ejecuta siempre en un solo ordenador (con varios núcleos o no).
- Se denomina «programación distribuida» a la creación de software que se ejecuta en ordenadores distintos, que hacen uso de su propia memoria local y que se comunican entre ellos a través de una red.

La programación distribuida se usa para tratar problemas computacionales complejos o masivos (con muchos datos). Permite por ello hacer grandes cálculos usando miles de ordenadores. Como ejemplo de ello fue el proyecto [SETI@home](#), que trataba de encontrar vida extraterrestre inteligente. Este proyecto fue apoyado por millones de personas que permitieron el uso de sus computadoras personales, que procesaban la información capturada por el radiotelescopio de Arecibo, emplazado en Puerto Rico.

Aplicaciones:

- Computación en la nube o servicios en la nube (Cloud Computing): Consiste poner a la disponibilidad de los usuarios varios servicios que no han de administrar ellos mismos, especialmente almacenamiento de datos y capacidad de cómputo.
- Las redes Peer-to-Peer Networks: Permiten el intercambio directo de información, en cualquier formato, entre los ordenadores interconectados. Este tipo de tecnología ha permitido abstraer aún más los sistemas distribuidos, de forma que,

para los usuarios externos tienen la sensación de estar utilizando una única máquina virtual. Como ejemplo tenemos el middleware P2P.

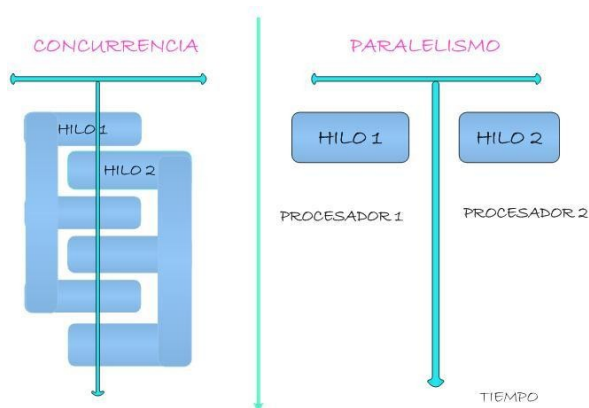


Figura 1: Programación concurrente / paralela
<https://java.codeandcoke.com/apuntes:concurrency>
 Autor: Santiago Faci

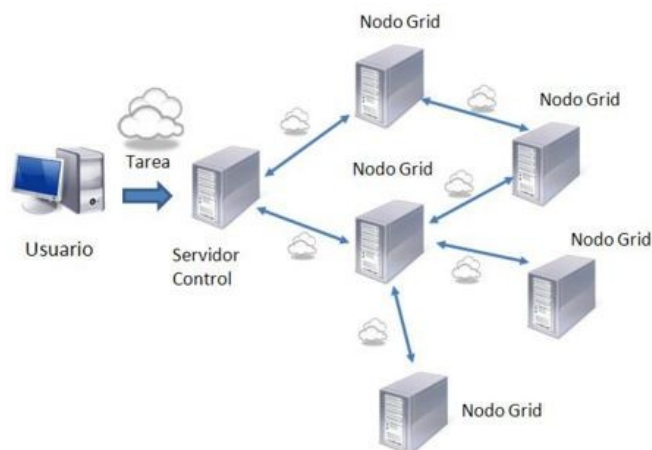


Figura 2: Programación distribuida
<https://java.codeandcoke.com/apuntes:concurrency>
 Autor: Santiago Faci

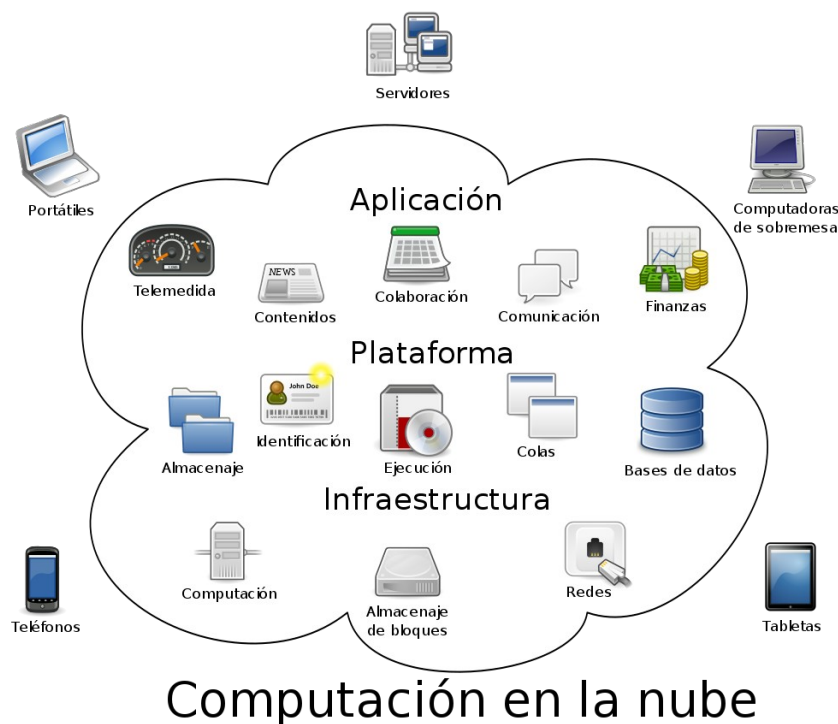


Figura 3: Programación distribuida
 De Sam Johnston, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=33656837>

2.3. HILOS

Un hilo o thread es cada una de las tareas que puede realizar de forma simultánea una aplicación. Por defecto, toda aplicación dispone de un único hilo de ejecución, al que se conoce como hilo principal. Si dicha aplicación no despliega ningún otro hilo, sólo será capaz de ejecutar una tarea al mismo tiempo en ese hilo principal.

Así, para cada tarea adicional que se quiera ejecutar en esa aplicación, se deberá lanzar un nuevo hilo o thread. Para ello, todos los lenguajes de programación, como Python o Java, disponen de una API o una librería para crear y trabajar con ellos.

En cualquier caso, es muy importante conocer los estados en los que se pueden encontrar un hilo. Estos estados se suelen representar mediante un gráfico como el que sigue:

✚ Estados de un hilo

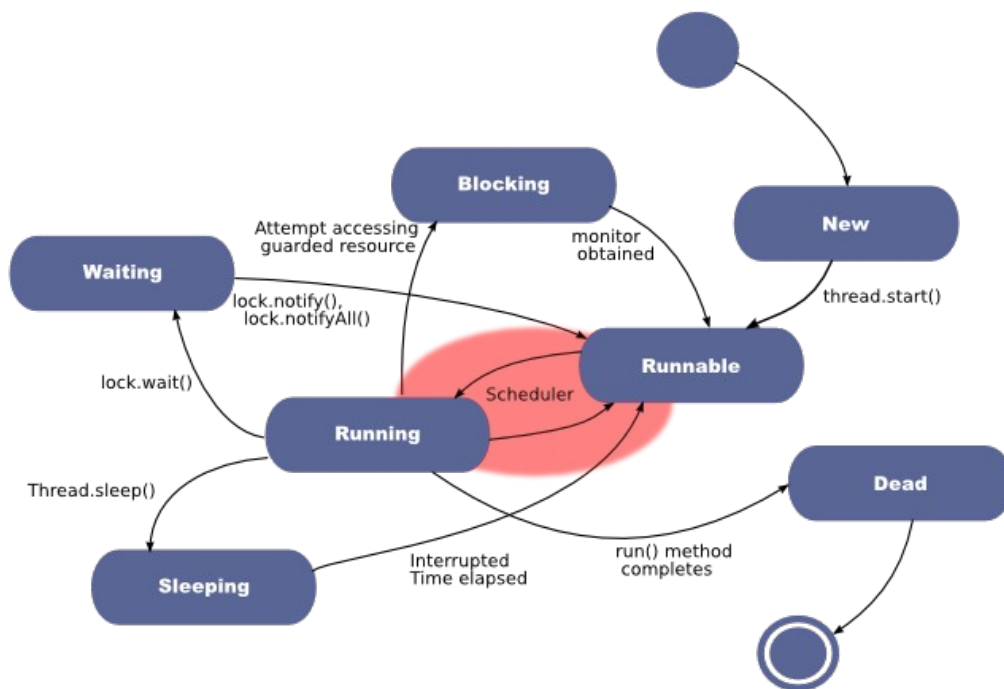


Figura 4: Estados de un hilo
<https://java.codeandcoke.com/apuntes/concurrencia>
Autor: Santiago Faci

3. ALGORITMOS DE PLANIFICACIÓN

En entornos multitarea, un algoritmo de planificación indica la forma en que el tiempo de procesamiento debe repartirse entre todas las tareas que deben ejecutarse en un momento determinado. Cada algoritmo planifica una política en la que define el orden de llegada de cada proceso, qué tarea se ejecuta primero, si tiene en cuenta prioridades, etc. Por ello existen diferentes algoritmos de planificación, cada uno con sus ventajas e inconvenientes, pero todos intentan cumplir con los siguientes puntos:

- Debe ser imparcial y eficiente
- Debe minimizar el tiempo de respuesta al usuario, sobre todo en aquellos procesos o tareas más interactivas
- Debe ejecutar el mayor número de procesos
- Debe mantener un equilibrio en el uso de los recursos del sistema

3.1. FCFS: First Come First Served

El **primer proceso** que llegue al procesador **se ejecuta** antes y **de forma completa**. Hasta que su ejecución no termine no podrá pasarse a ejecutar otro proceso. Funciona como una cola FIFO (First in - First out).

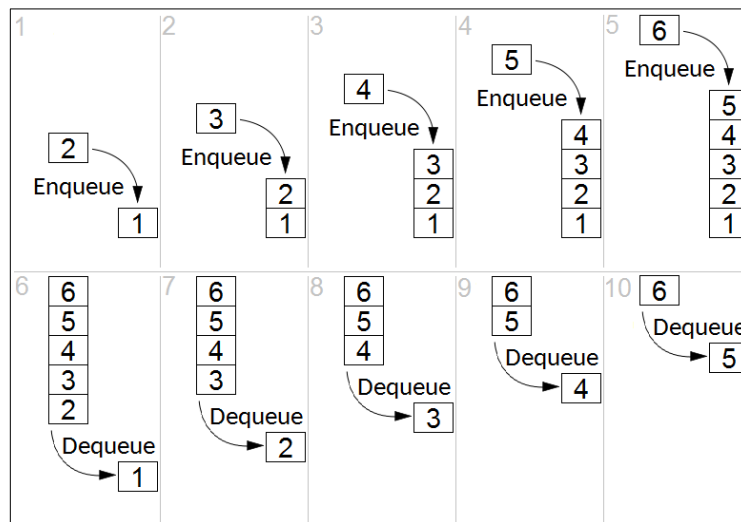


Figura 5: Ejemplo de cola FIFO

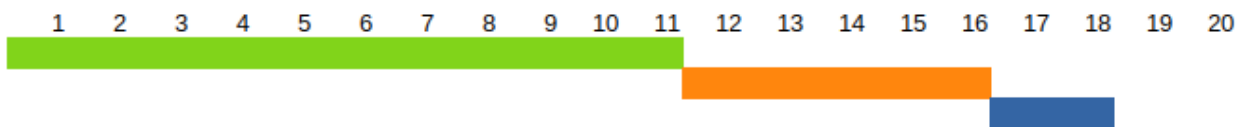
De Maxtremus - Own work, CC0, <https://commons.wikimedia.org/w/index.php?curid=44460399>

Este tipo de algoritmo es muy simple de implementar, pero no siempre resulta la forma más eficiente de tratar los procesos. Lo veremos en el siguiente ejercicio.

Ejercicio 1:

Tenemos los siguiente procesos (hay 3) y llegan a la CPU en el siguiente orden:

- **Proceso 1:** tiene un duración de 11
- **Proceso 2:** tiene una duración de 5
- **Proceso 3:** tiene una duración de 2



Vamos a calcular el tiempo de espera medio =

Suma de los tiempos de espera de cada proceso / el número de procesos

Número de procesos = 3

Tiempo de espera proceso 1 = **0** (no ha esperado)

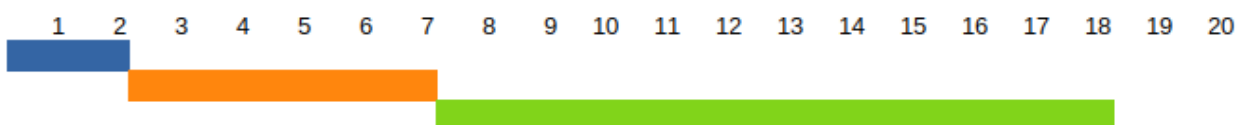
Tiempo de espera proceso 2 = **11** (ha esperado a que acabara el proceso 1)

Tiempo de espera proceso 3 = **16** (ha esperado a que acabara el proceso 2)

Tiempo de espera medio = $(0+11+16)/3 = 9$

Qué hubiera cambiado si los procesos hubieran entrado en otro orden. Primero el **azul**, luego el **naranja** y por último el **verde**.

- **Proceso 1:** tiene una duración de 2
- **Proceso 2:** tiene una duración de 5
- **Proceso 3:** tiene una duración de 11



Número de procesos = 3

Tiempo de espera proceso 1 = **0** (no ha esperado)

Tiempo de espera proceso 2 = **2** (ha esperado a que acabara el proceso 1)

Tiempo de espera proceso 3 = **7** (ha esperado a que acabara el proceso 2)

Tiempo de espera medio = $(0+2+7)/3 = 3$

Con esto vemos que el algoritmo puede ser bastante ineficiente. Depende mucho que el orden de entrada de los procesos sea "favorable".

3.2. SJF: shortes Job First

En este algoritmo, de todos los procesos listos para ser ejecutados, lo hará **primero el más corto**. Este algoritmo minimiza bastante el tiempo medio de espera.

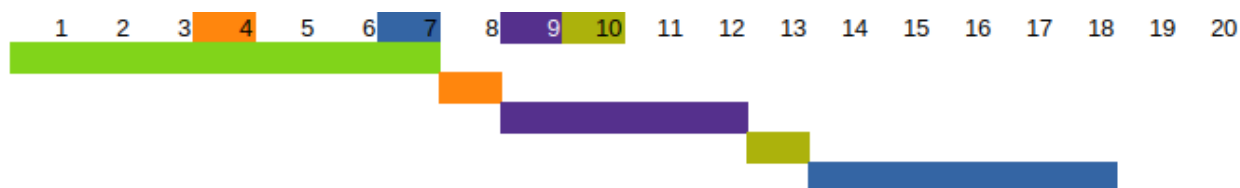
3.3. SRT: Shortest Remaining Time

De todos los **procesos** listos para ejecución, se ejecutará aquel al que **le quede menos tiempo para terminar**. **Es la versión expulsiva del algoritmo SJF.**

Ejercicio 2: Vamos a comparar SJF y SRT

Proceso	Llegada	Duración
Proceso 1	0 (llega el primero)	7
Proceso 2	3	1
Proceso 3	6	5
Proceso 4	8	4
Proceso 5	9	1

SJF



Número de procesos = 5

Tiempo de espera proceso 1 = 0 (no ha esperado)

Tiempo de espera proceso 2 = 4 (ha esperado a que acabara P1)

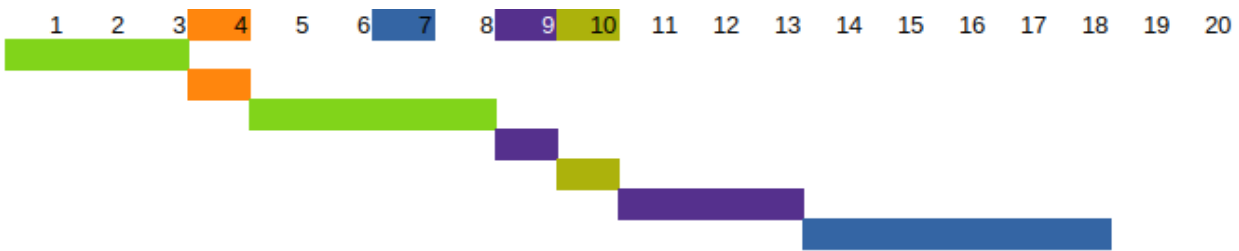
Tiempo de espera proceso 3 = 7 (ha esperado a que acabaran P1, P2, P4 y P5)

Tiempo de espera proceso 4 = 0 (no ha esperado, dura menos que P3)

Tiempo de espera proceso 5 = 3 (ha esperado a que acabe P4 y dura menos que P3)

Tiempo de espera medio = $(0+4+7+0+3)/5 = 2,8$

SRT



Número de procesos = 5

Tiempo de espera proceso 1 = 1 (le ha sacado P2 y ha tenido que esperar que cabe)

Tiempo de espera proceso 2 = 0 (ha esperado a que acabara P1)

Tiempo de espera proceso 3 = 7 (ha esperado a que acabaran P1, P2, P4 y P5)

Tiempo de espera proceso 4 = 1 (le ha sacado P5 y ha tenido que esperar que cabe)

Tiempo de espera proceso 5 = 0 (P4 le quedan 3 quantum y P5 tiene 1, entra P5)

Tiempo de espera medio = $(1+0+7+1+0)/5 = 1,8$

Podemos ver que SRT es más eficaz que SJF.

Ambos algoritmos tienen el mismo problema: **Riesgo de inanición.**

Es lo que le ha pasado al proceso P3, cada vez que podía ser ejecutado entraba o había en cola un proceso con menos duración que él. Eso podría acabar que P3 nunca se ejecuta. Una solución es poner prioridades y a medida que P3 vaya esperando (le pasa otro proceso por delante), aumentarle la prioridad, para que en un momento dado, aunque tenga más tiempo entra de todas formas.

3.4. RR: Round Robin

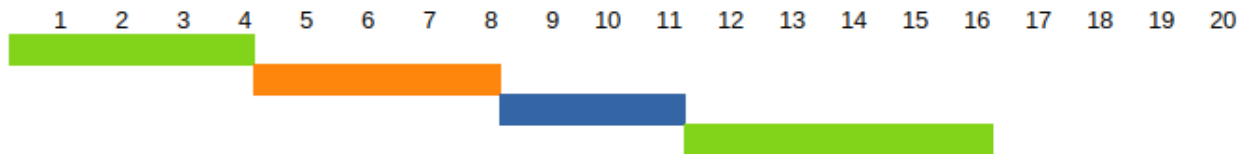
Se le conoce también como algoritmo de turno rotatorio. En este caso se designa una cantidad corta de tiempo (**quantum**) de procesamiento a todas las tareas. Las que necesiten más tiempo de proceso deberán esperar a que vuelva a ser su turno para seguir ejecutándose.

Ejercicio 3:

Para comprender mejor cómo funciona el quantum, haremos el siguiente ejercicio con $Q=4$, $Q=2$ y $Q=1$

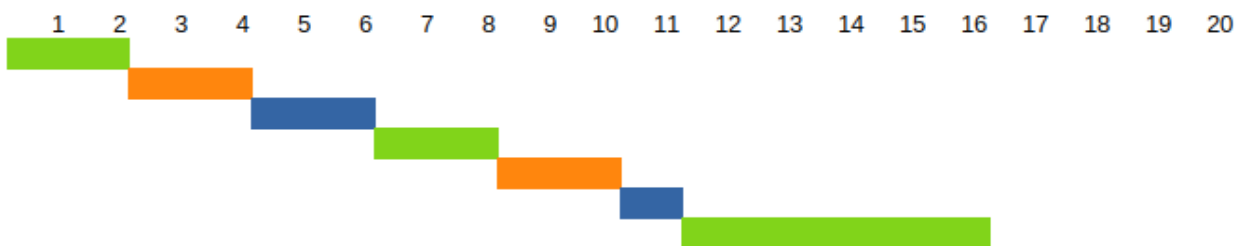
- **Proceso 1:** tiene una duración de 9
- **Proceso 2:** tiene una duración de 4
- **Proceso 3:** tiene una duración de 3

Q=4



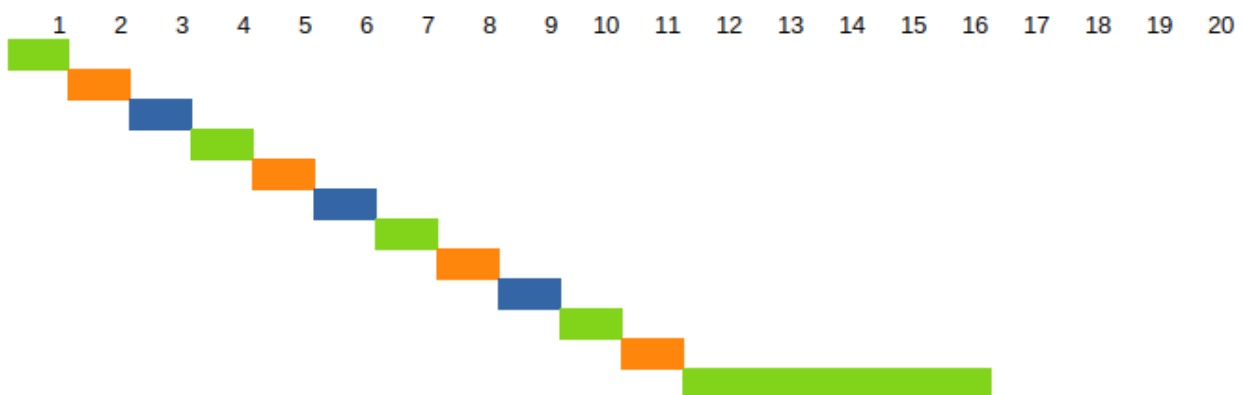
Tiempo de espera medio = $(7+4+8)/3 = 6,33$

Q=2



Tiempo de espera medio = $(7+6+8)/3 = 7$

Q=1



Tiempo de espera medio = $(7+7+6)/3 = 6,66$

Hay que tener en cuenta:

- Si Q es muy grande, en este ejemplo pongamos Q=10, se ejecutarían los procesos igual que en el algoritmo FCFS.
- Si Q es muy pequeño, en este ejemplo podríamos poner Q=0,5 o menos. Con tanto cambio de contexto (también tiene un coste de tiempo), hace que baje el rendimiento.

3.5. Varias colas con realimentación

Es un algoritmo más complejo que todos los anteriores y, por tanto, más realista. Se utiliza en entornos donde se desconoce el tiempo de ejecución de un proceso al inicio de su ejecución. En este caso, el sistema dispone de varias colas que a su vez pueden disponer de diferentes políticas unas de otras. Los procesos van pasando de una cola a otra hasta que terminan su ejecución. En algunos casos, el algoritmo puede adaptarse modificando el número de colas, su política, etc.

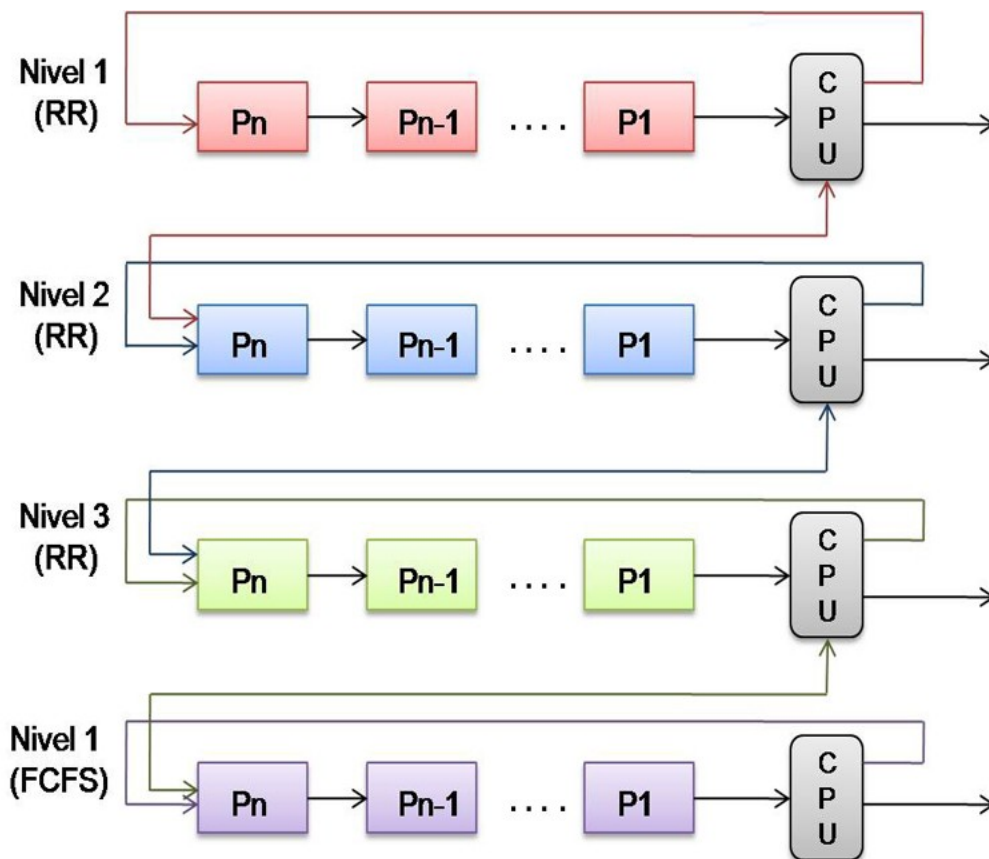


Figura 5: Estados de un hilo
<https://java.codeandcoke.com/apuntes:concurrency>
 Autor: Santiago Faci