Programación de Servicios y Procesos en Python

José Luis Carnero Sobrino

marcombo FORMACIÓN

PROGRAMACIÓN DE SERVICIOS Y PROCESOS EN PYTHON

José Luis Carnero Sobrino

PROGRAMACIÓN DE SERVICIOS Y PROCESOS EN PYTHON

José Luis Carnero Sobrino

Marcombo

Alpha Editorial S.A.
Calle 62 20-46 esquina, Bogotá
Teléfono (57-601) 746 0102
cliente@alpha-editorial.com
www.alpha-editorial.com

Libros digitales
wwww.alphaeditorialcloud.com

Programación de Servicios y Procesos en Python

Primera edición, 2022

www.marcombo.com

Diseño de la cubierta: ENEDENÚ DISEÑO GRÁFICO
Maquetación: cuantofalta.es
Corrección: Nuria Barroso y Mónica Muñoz
Directora de producción: M.ª Rosa Castillo

ISBN 978-958-778-852-5 (Colombia)
ISBN 978-958-778-853-2 (Digital)
ISBN 978-84-267-3477-8 (España)

Índice general

Unidad 1

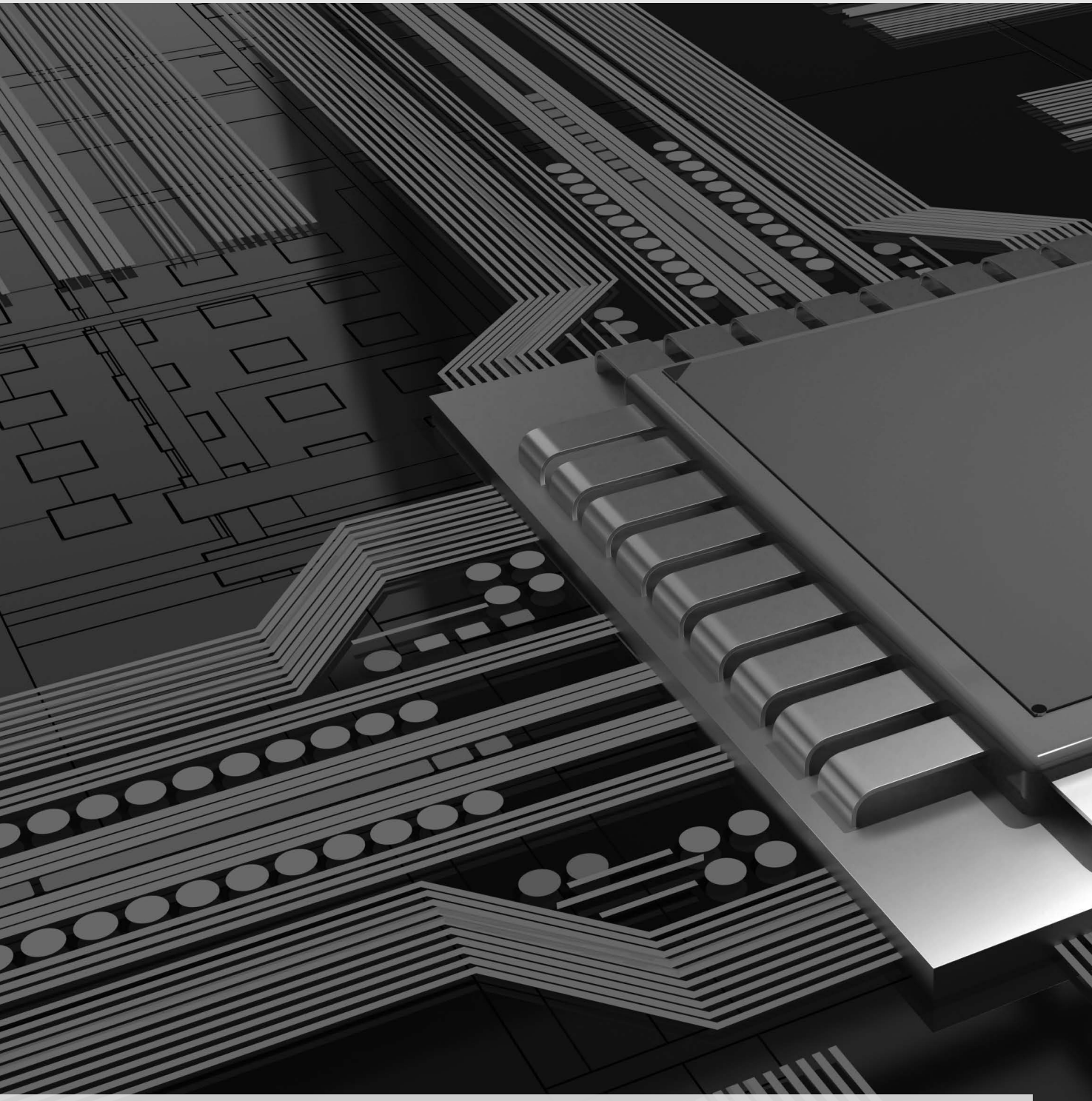
Unidad 2

Unidad 3

Unidad 4

Unidad 5

Unidad 1 Procesos



En esta unidad veremos:

1.1 Introducción y fundamentos

1.2 Concurrencia y distribución

1.3 Estados de un proceso

1.4 Gestión de procesos

1.5 Operaciones con procesos

1.1 Introducción y fundamentos

Los continuos avances en el hardware de los ordenadores requieren nuevas técnicas de software, empezando por los sistemas operativos, que permitan aprovechar el sistema de una forma más eficiente y con un mayor rendimiento en términos de consumo de tiempo y recursos.

Hay que empezar por pensar que, por cuestiones de velocidad, nuestro microprocesador cuenta únicamente con la memoria principal y con la memoria caché en términos de anticipación. De esta forma, cualquier programa y sus datos asociados deben trasvasarse entre la RAM y las memorias auxiliares (discos duros, memorias flash, almacenamiento en la red, etc.).

Podemos definir un **programa** como el conjunto de instrucciones que resuelven una tarea específica en un ordenador (microcontrolador, PC, smartphone, etc.). Hay que tener en cuenta que, además de las instrucciones, debemos tener asociado el espacio de memoria para hacer el correspondiente tratamiento de datos.

En la terminología moderna, a estos programas se los suele llamar **aplicaciones** o simplemente apps. También se puede considerar que una app puede estar compuesta por un conjunto de programas. En este caso, una aplicación es un software diseñado para resolver una función concreta de cara a un usuario.

Ejemplo 1

Si consideramos un procesador de texto como una aplicación, tendremos en cuenta que, además del programa principal que permite editar, tendremos muchos otros que, por ejemplo, revisen la ortografía, gestionen la impresión o hagan copias parciales de nuestros escritos en alguna zona de disco o de la nube.

Y, tras esta introducción, nos acercamos al concepto que más nos interesa. De forma sencilla, podemos decir que un **proceso** no es ni más ni menos que un programa en ejecución. Es la unidad básica que trata un sistema operativo y que consta de los siguientes elementos:

- Zonas de memoria en las que están alojadas las instrucciones y los datos.
- Conjunto de registros del microprocesador para poder efectuar un cambio de contexto, especialmente el registro contador que contiene la dirección de memoria de la siguiente instrucción para ejecutar.
- Estado del proceso que contiene la información necesaria para que el sistema operativo pueda ponerlo en ejecución en su correspondiente turno, si no está bloqueado por una operación de entrada/salida, etc.
- Otra información, como la identificación, procesos asociados, prioridades, estadística de uso, etc.

Ejemplo 2

En los ordenadores de sobremesa, casi todos los programas que lanzamos crean un proceso individual y por eso podemos tener cuatro ventanas distintas con Word, tres con el explorador de archivos, etc. En los smartphones la tendencia es la contraria: un solo proceso abierto por app.

En general, los programas pueden implementar al principio de su código la detección de algún proceso abierto de ese mismo programa o app, y decidir si crear un nuevo proceso o pasar a primer plano el proceso ya existente en función del sistema operativo o del programa en cuestión.

Por lo tanto, cada vez que el sistema operativo comienza la ejecución de un programa, este se carga o instancia en memoria, conforme al estándar dado en ese sistema. El mismo programa puede ejecutarse como dos o más procesos separados, con sus datos propios y su ejecución independiente.

Un **fichero ejecutable** es aquel que contiene la información necesaria para crear un proceso en memoria capaz de ser ejecutado por el procesador (figura 1.1).

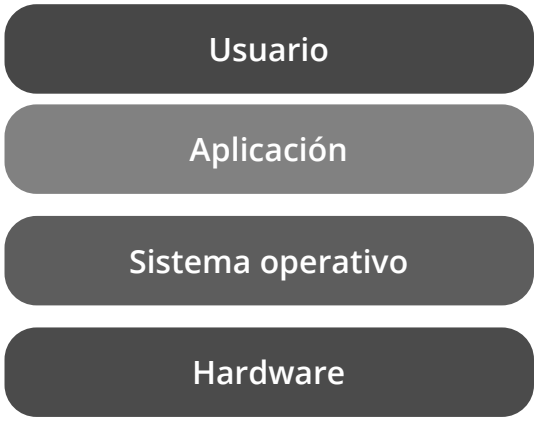


Figura 1.1
Capas de un sistema informático.

Los programas se pueden generar mediante:

- Compiladores, los cuales transforman código de algún lenguaje fuente en un fichero ejecutable por un sistema operativo; por ejemplo, C/C++.
- Intérpretes, que son programas que se encargan de transformar cada una de las líneas de un lenguaje de alto nivel en instrucciones máquina y ejecutarlas individualmente; por ejemplo, PHP.
- Técnicas híbridas, que compilan un lenguaje de alto nivel en otro intermedio, que posteriormente se interpretará por una máquina virtual con base en el sistema operativo; por ejemplo, Java o .net de Microsoft.

1.2 Concurrencia y distribución

Podemos imaginarnos, al principio de los tiempos de la informática, un ordenador que tuviese cargado un único programa y que se dedicase en exclusiva a ejecutar ese código hasta el momento en el que lo haya concluido, en cuyo caso podría pasar al siguiente programa. Este concepto se conoce como **sistema monoprogramación**. En la actualidad, todavía existen sistemas que se encargan del procesamiento por lotes o batch.

Posteriormente, y con el objetivo de simular una respuesta a múltiples tareas, surge el concepto de multiprogramación, por la cual dos o más procesos pueden coexistir en memoria y ejecutarse por turnos. Esto último se conoce como **programación concurrente** (figura 1.2).

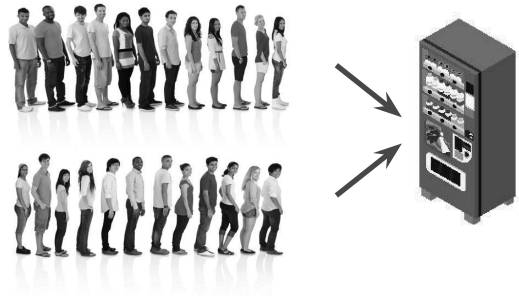


Figura 1.2
Programación concurrente.

Dos conceptos importantes en la programación concurrente son:

- El **quantum**. Es el tiempo límite que el sistema operativo asigna a la ejecución de un proceso, transcurrido el cual se pasa al siguiente.

- **El cambio de contexto**. Es el proceso llevado a cabo por el sistema operativo para salvaguardar los datos actuales de ejecución del proceso en curso y restaurar los del siguiente proceso que ejecutar.

De todo ello podemos deducir que la ejecución concurrente no produce mayor velocidad en el procesado de los programas; simplemente permite al ordenador lidiar con varias tareas (procesos) al mismo tiempo. Y, si la velocidad del procesador así lo permite, dar al usuario la sensación de que todo sucede a la vez: que, mientras escuchamos música y descargamos un fichero, podemos compilar un programa y manejar el procesador de textos, etc.

Hay que tener en cuenta que tiene que haber un buen equilibrio en el quantum. Por ponernos en los extremos:

- Si fuese demasiado grande, no percibiríamos correctamente la sensación de concurrencia.
- Si fuese extremadamente pequeño, perderíamos la mayor parte del tiempo en hacer cambios de contexto.

Si a todo lo anterior le añadimos que nuestro ordenador tiene varios procesadores o procesadores con diversos núcleos, entonces podríamos ejecutar distintas tareas al mismo tiempo en lo que se conoce como **programación paralela** (figura 1.3).

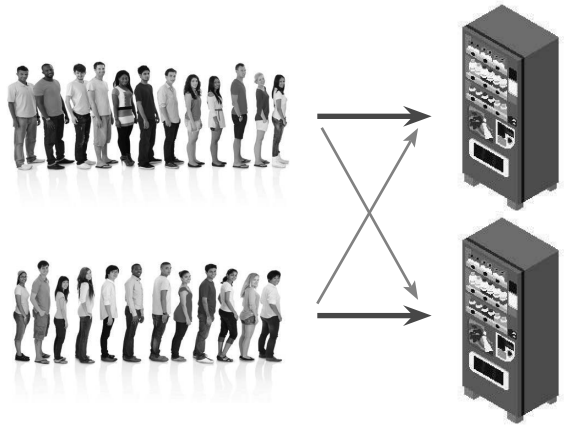


Figura 1.3
Programación paralela.

En este caso, las labores del sistema operativo se hacen más complejas, porque los recursos como la memoria siguen siendo compartidos, y hay procesos que son dependientes de otros y, por lo tanto, no se pueden ejecutar simultáneamente.

Ejemplo 3

Si imaginamos que tenemos una sola puerta de entrada a un recinto cultural, pero manejamos varias colas con sus prioridades (embarazadas, parejas, etc.), hablaríamos de **concurrencia**. Si además de tener varias colas, tenemos varias puertas de entrada y dinámicamente se puede redirigir, estaríamos hablando de **paralelismo**.

Tener varias líneas de ejecución no solo afecta al mejor rendimiento de ejecución de procesos, sino que un mismo proceso podría particionarse en líneas de ejecución para mejorar su rendimiento. Este particionamiento puede ser de dos formas:

- Manualmente, el programador puede codificar los hilos en los que un programa se podría dividir.
- Automáticamente, el compilador o la máquina virtual podrían generar hilos atendiendo a las sentencias que son linealmente independientes. Estas técnicas todavía están en vías de desarrollo.

Hasta ahora hemos pensado en la ejecución de los procesos en un único ordenador. Si pensamos en que se pudiera usar más de un ordenador para ejecutar nuestros procesos, haríamos alusión al concepto de **programación distribuida** (figura 1.4).

Figura 1.4
Programación distribuida.

Los ámbitos de aplicación del cómputo distribuido difieren de los anteriores y suelen ceñirse a cálculos aritméticos pesados y susceptibles de ser troceados para ejecutarse remotamente y, después, unificar su resultado. El impedimento en estos casos es que recursos como la memoria no son compartibles.

Ejemplo 4

Existe cierto tipo de malware llamado **cryptojacking**, que ejecuta en nuestro ordenador un proceso capaz de recibir datos a través de Internet, procesarlos y devolver resultados de forma continua, ocupando tiempo de cálculo de nuestro sistema (PC, smartphone, etc.). El objetivo de este virus puede ser minar distributivamente criptomonedas.

1.3 Estados de un proceso

Cuando arrancamos o creamos un proceso **nuevo**, el sistema operativo se toma un tiempo hasta que se ubican e inicializan todas las estructuras necesarias en la memoria; a esta estructura se la denomina **PCB** *(process control block)*.

Para saber más...

En la siguiente dirección se pueden consultar los componentes básicos de un PCB:

https://es.wikipedia.org/wiki/Bloque_de_control_del_proceso

Una vez que el proceso está completamente instanciado en memoria, pasa al estado de **listo** y, de manera sucesiva, obtendrá la atención del procesador, procesadores o núcleos a medida que se vaya dando tiempo o quantum a su **ejecución**.

Si entramos en una operación de entrada/salida (E/S) que no va a consumir tiempo de procesador, pasará a estar **bloqueado** hasta que complete su operación, momento en el que volverá a la cola de **procesos listos**. El sistema operativo podrá manejar colas para cada dispositivo de E/S y gestionarlas de forma independiente.

Si hemos completado la ejecución de un proceso y ha llegado a su fin, pasará al estado de **terminado**, momento en el que el sistema operativo libera los recursos asignados y queda a disposición de nuevos procesos.

Adicionalmente, podríamos definir otro estado, llamado **descargado** o **suspendido**, en el que, por falta u optimización de recursos, se guardaría temporalmente la estructura de un proceso en la memoria secundaria, dejando la memoria principal libre para otras operaciones (figura 1.5).

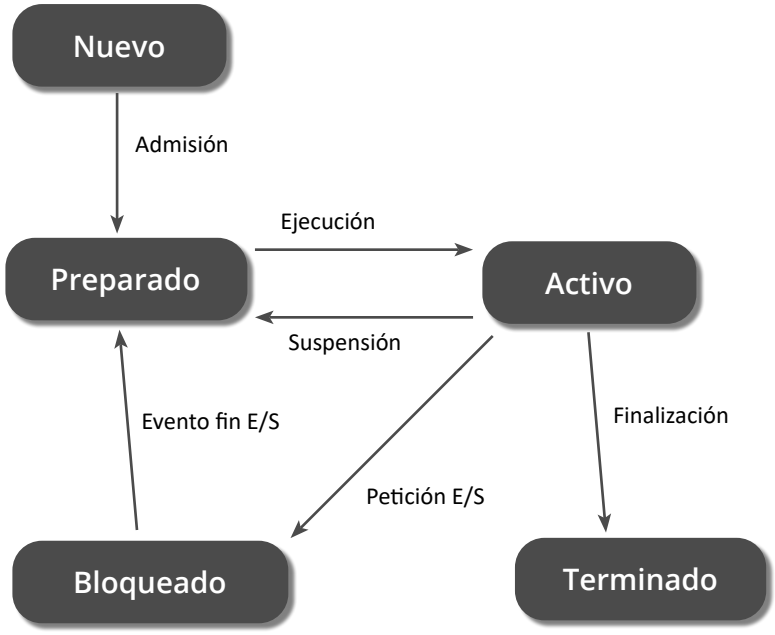


Figura 1.5
Estados de un proceso.

A continuación, enumeramos los estados más significativos de un proceso:

- **Nuevo**. Proceso en creación.
- **Listo**. Proceso a la espera de ejecución de sus instrucciones por parte de la CPU.
- **Ejecución**. Proceso cuyas instrucciones están siendo ejecutadas por la CPU.
- **Terminado**. Proceso que ha finalizado.
- **Bloqueado**. Proceso a la espera de una operación de E/S.
- **Suspendido**. Proceso que se ha guardado en una memoria secundaria (disco duro, etc.) para liberar recursos en la memoria principal (RAM).

1.4 Gestión de procesos

Como hemos dicho, los procesos son las unidades básicas de ejecución para el sistema operativo. Existe una tabla de PCB que el sistema operativo maneja para hacer el intercambio, asignar procesadores, bloquear recursos y zonas de memoria, controlar los estados, contabilizar los datos, etc.

Existen dos hardwares básicos para que todo el sistema de planificación e intercambio de procesos pueda funcionar de forma adecuada en un sistema de multiprogramación:

- Un mecanismo de **interrupciones**, mediante el cual una tarea o proceso puede detenerse temporalmente en el microprocesador, salvaguardando su estado y momento de ejecución para atender temporalmente a otra de mayor prioridad hasta que esta acaba. De forma recursiva cada tarea puede ser interrumpida por otra de mayor prioridad de forma recursiva. Hay que tener en cuenta que estas rutinas de tratamiento de interrupción deberían ser suficientemente rápidas en relación con el quantum establecido.
- Una protección de instrucciones críticas del procesador según el **modo usuario/supervisor**. El modo supervisor, kernel o privilegiado permite el uso de ciertos recursos e instrucciones restringidas de la CPU para hacer operaciones de interrupción, llamadas al sistema, cambio de contexto, etc. (figura 1.6).

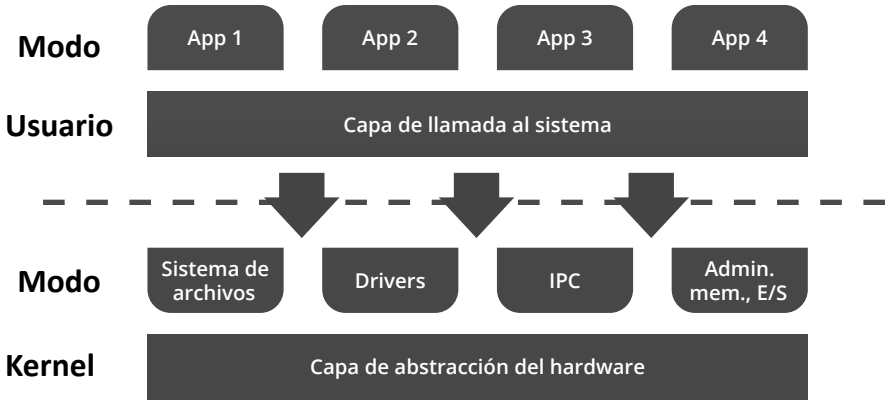


Figura 1.6
Funcionamiento modo usuario/supervisor.

Ejemplo 5

El sistema usa las interrupciones hardware para operaciones tan básicas como el manejo del ratón, ejecutando las rutinas asociadas a cada desplazamiento o pulsación con prioridad al resto de procesos de usuario.

Hay dos elementos en el núcleo o kernel del sistema operativo que tratan con los procesos: el **Dispatcher** y el **Scheduler**.

1.4.1 Dispatcher

El Dispatcher es el encargado de asignar a los procesos el uso de la CPU. Esto se ejecuta en colaboración con el Scheduler, para determinar cuál es el proceso al que se le debe dar entrada. Básicamente, se encarga de todo el trasiego de información entre memoria RAM y microprocesador, la actualización del PCB en función del cambio de estado, la planificación de los microprocesadores o núcleos del hardware a los cuales se le asigna un proceso, etc.

La ejecución del Dispatcher tiene lugar en alguna de las siguientes circunstancias:

- Se agota el quantum de tiempo asignado a un proceso.
- Un proceso termina su ejecución.
- Un proceso quiere desalojar el procesador por una espera programada, suspensión, sincronización o petición sobre algún recurso.
- Un proceso inicia una operación de E/S y se determina su espera hasta la finalización de dicha operación.
- Algún evento cambia un proceso de bloqueado a modo preparado.
- Existe alguna interrupción del sistema que desaloje temporalmente el proceso.

Cambio de contexto

El **cambio de contexto** es la función básica del Dispatcher y consiste en:

- Salvaguardar la situación hardware en la ejecución del proceso en curso, con su estado lógico actual, todo ello de manera que sea fácilmente susceptible de continuar en el punto exacto una vez restaurada su situación en próximos consumos de quantum.
- Cargar en el procesador la información de ejecución del nuevo proceso que se va a ejecutar.
- Establecer los valores nuevos en el PCB.

process control block

El PCB es una estructura de datos que contiene la información relativa a procesos como:

- Código de identificación del proceso.
- Nombre.
- Estado.
- Registros de CPU, con especial atención al PC (program counter).
- Prioridades que faciliten la labor del Scheduler.
- Punteros a las zonas de memoria manejadas (instrucciones y datos).
- Propietario y privilegios del sistema de recursos.
- Estado de las operaciones de E/S, etc.

1.4.2 Scheduler

El Scheduler es el encargado de hacer la planificación de los procesos, lo cual es especialmente relevante en los sistemas actuales que disponen de múltiples núcleos disponibles. Se encarga de gestionar las colas con las prioridades para tener a disposición del Dispatcher el siguiente proceso que debe atender la CPU.

El Scheduler tiene muchos modos de operar con base en múltiples criterios que van evolucionando con los sistemas operativos, las técnicas hardware para manejar los distintos procesadores y la combinación de técnicas de partición e hilos como hyper-threading. Con todo ello puede haber una cola única, colas por procesador y colas por niveles.

Para saber más...

Técnicas como el hyper-threading aumentan el rendimiento del sistema:

https://es.wikipedia.org/wiki/Hyper-Threading

Algunos de los criterios para hacer una planificación podrían ser: el tiempo de uso de la CPU, el rendimiento, el tiempo de respuesta o el número y nivel de núcleos. Con todo ello podemos enumerar a modo ilustrativo una serie de sistemas de planificación:

- First come first served (FCFS): se implementa una cola FIFO y los procesos se ejecutan en orden de llegada.
- Shorted job first (SJF): se da prioridad a los procesos que tengan un menor tiempo de ejecución.
- Basados en prioridad: a cada proceso se le da una prioridad que puede ser dinámica para evitar la postergación indefinida.
- Round-robin: a cada proceso se le da un quantum y, al acabar, vuelve al final de la cola. A mayor quantum, mayor tiempo de ejecución y, por lo tanto, más rápido.
- Cola multinivel: se puede establecer una cola para cada conjunto de cualidades de procesos o procesadores que utilizar y ordenarlas de forma independiente.

Ejercicio resuelto

Simular el comportamiento de un sistema y los estados de los procesos a lo largo del tiempo en un ordenador ficticio configurado con un quantum de 100 ciclos de reloj y un cambio de contexto de 10 ciclos de reloj. Se dispone de los siguientes procesos:

- Proceso 1: longitud de 400 instrucciones.
- Proceso 2: longitud de 450 instrucciones y una operación de E/S al disco duro en la instrucción 140 y con una duración de 300 ciclos.
- Proceso 3: longitud de 300 instrucciones y con una operación de transmisión de su búfer de datos a través de la red en la instrucción 120 y con una duración de 500 ciclos.
- Proceso 4: longitud de 250 ciclos, y terminación por un error o excepción en la instrucción 240.

Nota: Vamos a simplificar nuestro sistema suponiendo lo siguiente:

- que cada instrucción se ejecuta en un ciclo de reloj;
- que se puede predecir la duración de las operaciones de E/S en tiempo de procesador (ciclos de reloj);
- que las colas son FIFO y, cuando un proceso pasa a listo, lo hace en última posición;
- que todos los procesos están inicialmente listos, y la cola ordenada de P1 a P4.

Solución

CC = cambio de contexto

Entre paréntesis se muestra la instrucción o programa contador de un proceso al abandonar el procesador.

Tiempo	Suceso
0	P1 pasa Ejecución
100	P1 pasa a Listo (101). Empieza el CC
110	Culmina el CC y P2 pasa a Ejecución
210	P2 pasa a Listo (101). Inicio CC
220	Fin CC y P3 pasa a Ejecución
320	P3 pasa a Listo (101). Inicio CC
330	Fin CC y P4 pasa a Ejecución
440	P4 pasa a Listo (101). Inicio CC
450	Fin CC. P1 pasa a Ejecución
550	P1 pasa a Listo (201). Inicio CC
560	Fin CC. P2 pasa a Ejecución
600	P2 (140) pasa a Bloqueado E/S Disco. Inicio CC
610	Fin CC. P3 pasa a Ejecución
620	P3 (120) pasa a bloqueado E/S Red. Inicio CC

Tiempo	Suceso
630	Fin CC. P4 pasa a Ejecución
730	P4 pasa a Listo (201). Inicio CC
740	Fin CC. P1 pasa a Ejecución
840	P1 pasa a Listo (301). Inicio CC
850	Fin CC. P4 pasa a Ejecución
890	P4 (240) pasa a Terminado con código de error. Inicio CC
900	Fin CC. P1 pasa a Ejecución
900	P2 termina E/S y pasa a Listo
1000	P1 (400) pasa a Terminado. Inicio CC
1010	Fin CC. P2 pasa a Ejecución
1110	P2 (251) sigue, consume otro quantum, pues no hay procesos en cola de Listos
1120	P3 termina E/S red y pasa a Listo
1210	P2 (351) pasa a Listo. Inicio CC
1220	Fin CC. P3 pasa a Ejecución
1320	P3 (221) pasa a Listo. Inicio CC
1330	Fin CC. P2 pasa a Ejecución
1430	P2 (450) pasa a Terminado. Inicio CC
1440	Fin CC. P3 pasa a Ejecución
1420	P3 (300) pasa a Terminado

Ejercicio propuesto 1

Haga una simulación similar imaginando que disponemos de dos procesadores o núcleos. ¿Es exactamente el doble de rápido?

Vocabulario

Ciclo de reloj: pulso electrónico sincronizado por un oscilador que coordina el funcionamiento de la CPU en sus órdenes e instrucciones internas. La velocidad de un microprocesador se mide en megahercios (MHz), gigahercios (GHz), terahercios (THz), etc.

1.5 Operaciones con procesos

Ya hemos visto que un proceso es la unidad básica de ejecución para el sistema operativo, y que en un sistema multiprogramación pueden convivir varios al mismo tiempo. Los procesos se crean y pasan por distintos estados, desapareciendo en el caso de que lleguen a la situación de "terminado". Un proceso puede crearse de forma interactiva por un usuario (haciendo doble clic en el icono de un programa, tecleando un comando, etc.) o ser el sistema operativo u otro proceso en ejecución los que inicien un nuevo proceso.

También hay que tener en cuenta que, en el proceso de arranque o inicio del sistema operativo, se inician un número considerable de procesos de sistema que se encargan de las funciones básicas tales como privilegios, sistema de ficheros, servicios básicos de red o interfaz de usuario, por poner algunos ejemplos.

1.5.1 Clonación de un proceso: fork

Una forma de crear un nuevo proceso es a través de la conocida función *fork()*, implementada en muchas librerías y típicamente ligadas al mundo unix/Linux. Mediante esta función, un proceso era capaz de replicarse en memoria a través de la operación **fork**. Esta función obtendrá un código de vuelta que nos indicaría a partir de la invocación si tratábamos con el proceso original o el proceso hijo. Esta es una forma un tanto romántica, y algo obsoleta, dado que la porción de código previa a la ejecución del fork se duplica en los dos procesos con escasa probabilidad de ejecutarse de nuevo. Esto da lugar, en la mayoría de los casos, a un consumo extra del recurso de espacio en memoria.

Para saber más...

Todos los ejemplos de esta unidad se encuentran en www.marcombo.info, en la carpeta denominada **Procesos**. Código de acceso: MARCOMBO9.

Ilustramos el listado 1.1 a modo de ejemplo anecdótico para la creación de un proceso mediante fork, aunque de manera más correcta podríamos llamarlo "clonación de procesos". Tenemos que matizar que esta función está disponible únicamente en sistemas Unix/Linux.

La función **fork** de la librería os tiene la función de replicar un proceso en el sistema con todas sus singularidades, tales como alojar los datos e instrucciones en memoria o crear el nuevo elemento en la tabla de procesos con todas sus características de estado, identificador, etc. A partir de ahí, los dos procesos están listos para ser ejecutados en el momento en el que el microprocesador o los diferentes núcleos puedan concederle un quantum bajo los designios del sistema operativo.

La ejecución de cada uno de los procesos nuevos creados se inicia en la línea siguiente a la invocación a la función fork. Este es un buen momento para identificar si el proceso que estamos ejecutando es el original o el duplicado. En el ejemplo del listado "if newpid ==0".

A cada uno de los dos procesos (el padre que ya existía y el hijo que ha sido creado) la función fork les devuelve un valor diferente:

- Al padre le devuelve el identificador PID del hijo creado. Esto le concede su referencia de forma directa para un posible uso posterior.
- Al hijo le devuelve el valor cero, con lo cual no tiene referencia de su padre, aunque este valor podría ser almacenado en alguna variable como paso previo a la clonación.

Listado 1.1 Clonación de procesos mediante fork()

```
# fork solo funciona en unix/macos
import os

def padre():
   while True:
      newpid = os.fork()
      if newpid == 0:
         hijo()
      else:
         pids = (os.getpid(), newpid)
         print("Padre: %d, Hijo: %d\n" % pids)
      reply = input("Pulsa 's' si quieres crear un nuevo proceso")
      if reply != 's':
         break

def hijo():
    print('\n>>>>>>>>>> Nuevo hijo creado con el pid %d a punto de finalizar<<<<<' %
 os.getpid())
   os._exit(0)

padre()
```

Vocabulario

PID: *process identifier* es un número único que el SO asigna a cada uno de los procesos.

En nuestro programa podemos ver que la función padre() lanza un bucle infinito en el que clona el proceso actual a través de fork(), mientras el usuario decida seguir haciéndolo (a través del carácter "s"). Lo que hacemos en la función hijo() es finalizar el proceso actual (el recientemente creado); de lo contrario, por cada vez que clonásemos un proceso, tendríamos dos nuevos procesos preguntando si deseamos seguir replicándolos.

En el listado 1.2 podemos observar cómo se podría simular la operación fork() a través de un **Process**, dado que para sistemas Windows, por ejemplo, el fork no es operativo.

Listado 1.2 Simulación de fork con Process

```
from multiprocessing import Process
import os

def hijo():
   print("Padre: %d, Hijo: %d\n" % ( os.getppid(),os.getpid()))
   os._exit(0)

def padre():
  while True:
    p = Process(target=hijo)
    p.start()
    print ("\nNuevo hijo creado " , p.pid)
    p.join()
    reply = input("Pulsa 's' si quieres crear un nuevo proceso\n")
    if reply != 's':
      break
if __name__ == '__main__':
  padre()
```

A continuación, vamos a abordar la creación y manipulación de un proceso a través de las líneas de código de otro proceso en ejecución, como una labor del programador. Usaremos para ello el módulo **subprocess**, que se encuentra en la librería **Lib/subprocess.py**.

Para saber más...

Módulo subprocess en Python:
https://docs.python.org/3/library/subprocess.html

1.5.2 Creación y ejecución

La forma más sencilla de lanzar un proceso sería usar el método estático **run** de un subprocess, al cual de forma básica podemos pasarle:

- La ruta completa de un fichero ejecutable (.exe).
- El nombre de un ejecutable que pueda localizarse a través del path del sistema y que puede completarse con la extensión .exe en caso de no indicarla.
- El ejecutable con su lista de parámetros, con lo cual podríamos, por ejemplo, lanzar un editor y pasarle el fichero a ser abierto.

Para recordar...

En la actualidad, se pueden crear procesos indistintamente con "call" o "run", si bien a partir de la versión 3.5 se recomienda el uso de "run" por la posibilidad de que "call" pase a estar *deprecated* ("obsoleto") en futuras versiones.

Listado 1.3 Creación simple de procesos

```
import subprocess
try:
  subprocess.run(['Notepad.exe',])
  subprocess.run(['c:/windows/notepad.exe',])
  subprocess.run(['Notepad.exe','texto.txt'])
except subprocess.CalledProcessError as e:
    print(e.output)
```

En el listado 1.3 hay que tener en cuenta que, en caso de no poder crear un proceso, por ejemplo, de no existir el fichero o programa asociado, retornaría un valor **null** y daría una excepción que fácilmente se puede capturar en un try…catch.

Ejercicio propuesto 2

Lanzar un proceso que abra el navegador con la búsqueda en Google de un término solicitado al usuario.

Si queremos enviar parámetros al proceso para ser ejecutado, estos deberán ser enumerados en la lista, de forma posterior al nombre del proceso. En el listado 1.4, podemos ver cómo hacer cinco pings consecutivos, enviando los argumentos correspondientes, para comprobar la conectividad con un URL dado.