
Perspectiva histórica del procesamiento por lotes

PID_00276657

Joan Tomàs Matamalas Llodrà

Joan Tomàs Matamalas Llodrà

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Albert Solé Ribalta

Primera edición: marzo 2021
© de esta edición, Fundació Universitat Oberta de Catalunya (FUOC)
Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Joan Tomàs Matamalas Llodrà
Producción: FUOC
Todos los derechos reservados

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea este eléctrico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita del titular de los derechos.

Índice

Introducción.....	5
1. Perspectiva histórica.....	7
1.1. Los primeros ordenadores	7
1.2. <i>Mainframes</i> y procesamiento por lotes	8
1.3. Sistemas de tiempo compartido	10
1.4. Ordenadores personales	13
1.5. El procesamiento por lotes en la actualidad	14
2. Sistemas de procesamiento por lotes en entornos UNIX.....	15
2.1. Introducción	15
2.2. Automatización de la ejecución	15
2.2.1. Comando <i>nice</i>	15
2.2.2. Comandos <i>at</i> , <i>atq</i> , <i>atrm</i> y <i>batch</i>	16
2.2.3. Comando <i>cron</i>	18
3. Clústeres de computación: sistemas de colas.....	20
3.1. Arquitectura	20
3.2. Sistema de colas	22
3.3. Diferencias con los sistemas <i>big data</i>	24
4. Introducción a los sistemas UNIX.....	26
4.1. Sistema de ficheros	27
4.1.1. Estructura	27
4.1.2. Interacción	28
4.2. Breve introducción al <i>bash scripting</i>	31
Bibliografía.....	35

Introducción

Las técnicas que veremos en esta asignatura están relacionadas con el procesamiento de cantidades ingentes de datos. Para aplicarlas, se requiere de entornos de computación distribuidos en los que cada una de las máquinas analiza una parte de los datos y, posteriormente, los resultados de cada una de las máquinas se agregan para retornar el resultado del análisis deseado. Este avanzado sistema de procesamiento no surge de la nada, es el resultado de evoluciones tecnológicas tanto en la capacidad computacional como en la captura y el almacenamiento de datos.

Como en muchas facetas de la vida, saber de dónde venimos es esencial para entender dónde estamos y hacia dónde vamos. El objetivo de este primer módulo es recorrer de forma cronológica la historia de la computación y, por ende, del procesamiento de datos, abarcando desde las máquinas de tubos de vacío hasta los superordenadores. Veremos sus características, limitaciones y cómo en cada uno de ellos se organizaba el análisis de datos. Además, se introducirán en detalle dos de los sistemas de procesamiento en modo *batch* más utilizados hoy en día: el procesamiento por lotes en entornos UNIX y la gestión de colas de ejecución en entornos de computación de alto rendimiento.

1. Perspectiva histórica

Resultaría imposible definir una separación temporal que clasificara de manera clara las diferentes etapas evolutivas de la computación. Dependiendo del objetivo se podría clasificar por tecnología utilizada, potencia de cálculo, casos de uso, etc. En este módulo vamos a basarnos en la clasificación propuesta por Andrew Tanenbaum (2015) en su ya clásico *Modern operating systems*. Hemos seleccionado esta clasificación porque, a nuestro juicio, captura de manera clara las transiciones observadas en la gestión de procesos: de los primeros ordenadores, al procesamiento por lotes, siguiendo con las máquinas multiusuario y finalizando con los ordenadores personales y la gestión multitarea.

1.1. Los primeros ordenadores

Aunque resulta complejo definir cuál fue el primer ordenador moderno, no todas sus características aparecieron de repente, podemos datar su creación en la década de los cuarenta del siglo XX¹.

⁽¹⁾Obviando la máquina analítica (1837) propuesta por Charles Babbage.

El primer ordenador eléctrico programable de uso específico, Colossus, fue creado en 1943 en Bletchley Park (Reino Unido), el principal centro de desciframiento británico durante la Segunda Guerra Mundial. Este estaba construido con tubos de vacío y su principal finalidad era participar en el desciframiento de mensajes de las potencias del eje.

Puede considerarse el ENIAC (EE. UU., 1946) como el primer ordenador digital de propósito general. Su tarea principal era calcular trayectorias balísticas y su creación supuso reducir drásticamente el coste de estas operaciones, dado que era capaz de ejecutar tareas que requerían veinte horas de trabajo en tan solo treinta segundos. Dada su flexibilidad, el ejército estadounidense extendió su uso a diferentes problemas, entre ellos los cálculos que llevaron a la creación de la bomba de hidrógeno.

Aun así, el principal problema de estos ordenadores y sus contemporáneos, como el Mark 1 o el UNIVAC, era la manera de interactuar con ellos. Sin la existencia de lenguajes de programación, la única manera que estas máquinas podían ser programadas era mediante **complejos tableros de conexiones manejados por expertos operarios**, los primeros programadores. En la figura 1 podéis ver cómo eran esos tableros. Vamos a revisar lo que suponía para un ingeniero de la época utilizar estas máquinas:

1) Primero tenía que escribir una solicitud especificando qué tipo de programa quería ejecutar y cuánto tiempo de computación requeriría.

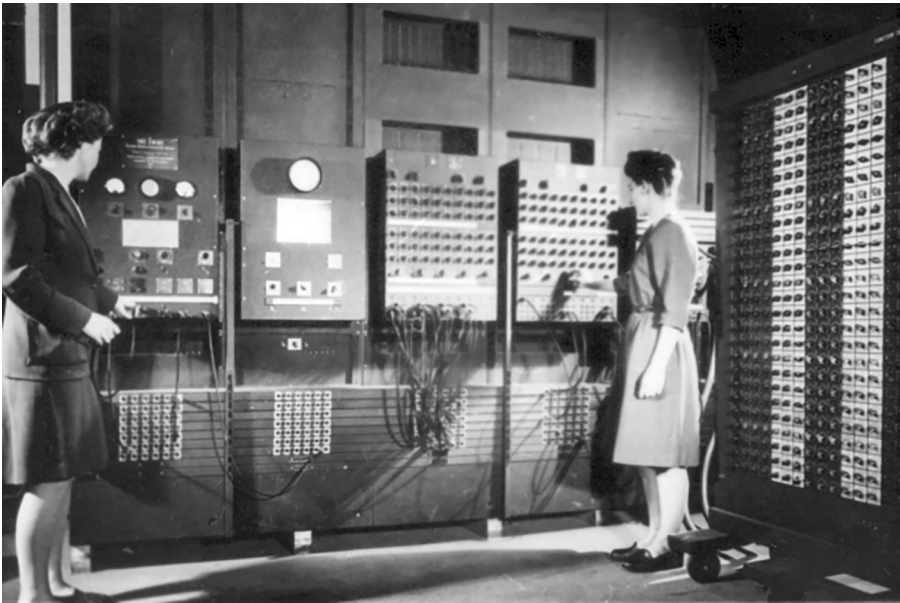
2) En el caso de que fuera aprobado, tenía que esperar un tiempo considerable a que llegara su turno, dada la limitación de recursos.

3) Una vez había llegado su *slot* temporal de uso de la máquina, tenía que ir al centro de computación y, con la ayuda de programadores, introducir su rutina en la máquina.

4) Si era afortunado y no había ningún error en su rutina ni ningún fallo mecánico, entonces podía obtener los resultados.

Todo esto cambió en la segunda generación de ordenadores.

Figura 1. Programadoras operando el panel de control del ENIAC en la escuela de ingeniería eléctrica Moore (U.S. Army)

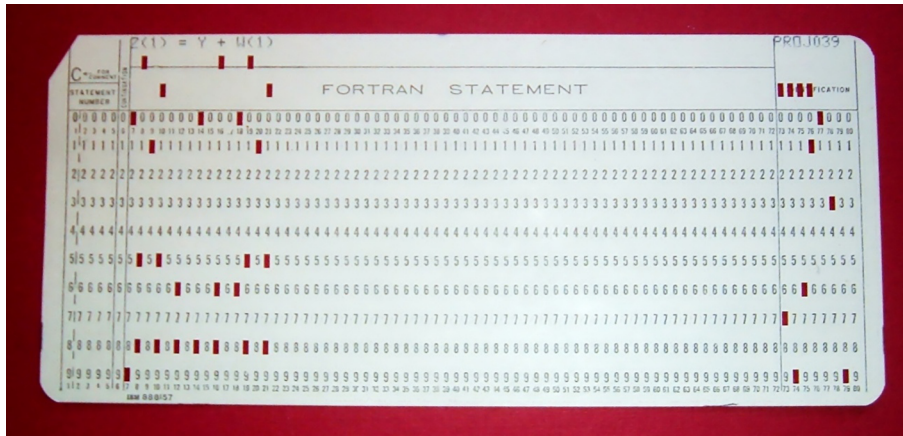


1.2. *Mainframes* y procesamiento por lotes

En la década de los cincuenta del siglo pasado, dos grandes avances popularizan enormemente el uso de los ordenadores:

- La irrupción de los transistores, que permiten reducir el tamaño y aumentar la capacidad de cómputo.
- El uso de tarjetas perforadas (figura 2) en sustitución de los paneles de control.

Estas máquinas dejan de ser de uso exclusivo de las organizaciones gubernamentales y empiezan a ser adoptadas por grandes corporaciones con finalidades comerciales. Estos nuevos sistemas de computación son los llamados *mainframes*.

Figura 2. Tarjeta perforada con la instrucción codificada en Fortran $Z(1) = Y + W(1)$ 

En esta generación de computadores es donde aparecen los primeros lenguajes de programación, como Fortran, contribuyendo también a la popularización de los ordenadores.

El uso de los *mainframes* supuso un gran aumento de la productividad, ya que los programadores dejaron de dedicar numerosas horas a la traducción de sus rutinas a lenguaje máquina. Además, al trabajar directamente con tarjetas perforadas, los flujos de cómputo cambiaron radicalmente, ya que ahora las diferentes tareas podían distribuirse de manera más eficiente:

- 1) El programador escribía su rutina en una hoja de papel en un determinado lenguaje de programación.
- 2) Después, utilizando una tabla de equivalencias, trasladaba las instrucciones a una tarjeta perforada.
- 3) Una vez completado el programa, el programador llevaba estas tarjetas al cuarto de cómputo, donde se asignaba una prioridad de ejecución.
- 4) Allí, un operario, siguiendo esta prioridad, cogía individualmente cada una de las rutinas, introducía las tarjetas en la máquina, recogía su resultado y proseguía con el siguiente programa de la lista.

Este modo de ejecutar los programas pronto supuso un gran problema. Con el avance en potencial de cálculo y el coste de los ordenadores, todo el tiempo empleado por los operarios en el cambio manual de programas suponía al cabo del año un volumen considerable de ciclos de computación perdidos. Además, el número de tareas que las empresas derivaban a operaciones computacionales era cada vez más grande, lo que complicaba notablemente la gestión individual de cada una de esas tareas. Con el objetivo de optimizar todo el proceso, aparece el **procesamiento por lotes**.

El procesamiento por lotes supuso una revolución en la computación y, de hecho, hizo que aún más empresas decidieran informatizar sus organizaciones. En este nuevo paradigma:

- 1) Los programadores llevaban sus tarjetas perforadas al centro de cómputo donde, en vez de procesarse individualmente, eran apiladas junto con otras rutinas pendientes de procesar.
- 2) Cuando el volumen de tarjetas era suficientemente grande, estas se transcribían a un nuevo dispositivo: la cinta magnética.
- 3) Esta cinta magnética era trasladada al ordenador de cómputo, donde las rutinas eran ejecutadas secuencialmente una detrás de otra, gravando cada uno de los resultados en otra cinta magnética para su posterior análisis.

Esto redujo los tiempos muertos entre ejecuciones consecutivas de las rutinas (ya no debíamos cargar nuevas tarjetas manualmente) a uno, cada vez que debíamos cambiar de cinta. Dado que las cintas podían guardar miles de rutinas, este nuevo sistema supuso un incremento sin parangón en la productividad. De hecho, este paradigma de procesamiento es tan popular, que incluso hoy podemos encontrarlo en nuestros ordenadores.

Ved también

En el apartado 2 de este módulo veremos más sobre el procesamiento por lotes en los ordenadores actuales.

1.3. Sistemas de tiempo compartido

Aún con todas las ventajas que supuso la aparición del procesamiento por lotes, las máquinas de cómputo seguían procesando trabajos secuencialmente, programa tras programa. Esto significaba que todos los dispositivos de la máquina estaban destinados a un trabajo determinado se usaran o no.

Ejemplo: la CPU y los dispositivos de entrada y salida

Trabajos con alta carga de cálculo, como simulaciones científicas o proyecciones militares, utilizaban mayoritariamente la CPU dejando los dispositivos de entrada y salida, como el lector de cintas magnéticas, en desuso. Por el contrario, muchas aplicaciones comerciales, como el cálculo de *stocks*, utilizaban mayormente el dispositivo de entrada y salida casi sin utilizar la CPU. Esto suponía un desaprovechamiento considerable de los recursos del sistema.

Además, así como la tecnología informática fue penetrando en más ámbitos de la sociedad, las grandes compañías productoras de ordenadores diversificaron su oferta para adaptarse a las diferentes necesidades de los consumidores. Desde grandes instituciones gubernamentales que requerían gran capacidad de cómputo, hasta empresas de tamaño medio que deseaban informatizar parte de sus procesos con sistemas más pequeños destinados solamente a analizar sus datos. Esta diversificación introdujo un problema adicional. Hasta ese momento cada máquina funcionaba con un primitivo sistema operativo específico para cada arquitectura de hardware. Esto suponía mantener y actualizar este software para cada una de las arquitecturas, lo que restringía notablemente la variabilidad de máquinas que se podía ofrecer.

Todo esto cambió en 1964 con la aparición del *mainframe* **IBM System/360** (figura 3), que supuso toda una revolución en el campo de la informática. Para empezar, fue el primero en utilizar circuitos integrados en lugar de transistores independientes, lo que abarató los costes y redujo significativamente el tamaño de los ordenadores. Gracias a la utilización de una arquitectura unificada y a la utilización de un sistema operativo más moderno, el OS/360, el IBM System/360 se ofrecía en una amplia variedad de modelos que iban desde los más económicos como el modelo 30, capaz de realizar 34.500 operaciones por segundo, al más avanzado modelo 91, con una capacidad de cálculo de 16,6 millones de operaciones por segundo.

Figura 3. *Mainframe* IBM System/360 Model 20 (izquierda) con lector de tarjetas multifunción IBM 2560 (derecha)



El IBM System/360 incluía otra característica típica de los ordenadores de esta generación: el **particionado de memoria**. Gracias a esta tecnología, la máquina era capaz de almacenar en la memoria diferentes programas. Los nuevos sistemas operativos, como el OS/360, se encargaban de gestionar los recursos del sistema optimizando el uso de todas sus partes.

A partir de ese momento, la máquina podía tener programas de procesamiento de datos que hacían uso del sistema de entrada y salida mientras, al mismo tiempo, se ejecutaban otras tareas que requerían un uso más intensivo de la CPU, como el cálculo de modelos de predicción.

Además, estos sistemas incluían una nueva técnica, el *spooling*, que eliminaba la necesidad de introducir los programas en el ordenador en grupos. Con esta nueva tecnología:

1) Los programadores simplemente llevaban sus tarjetas perforadas a la habitación de computación donde quedaban apiladas en un dispositivo de entrada (como el IBM 250 de la figura 3).

2) El sistema iba leyendo estas tarjetas según las iba introduciendo en la memoria (figura 3).

Esto optimizó aún más los ciclos de computación permitiendo un uso ininterrumpido de la CPU.

La introducción de todas estas innovaciones tecnológicas propició un **auge en la expansión de los sistemas informáticos**, pero no resolvió todos los problemas. Los programadores cada vez tenían que informatizar procesos más complejos, pero esto suponía un problema.

Ejemplo: el programador y los errores

Imaginad lo que suponía para un programador trabajar en un algoritmo donde cada vez que quería probar alguna de sus características tenía que transcribirlo a unas tarjetas perforadas y ponerse en la cola detrás de procesos que podían tardar horas en ejecutarse. Si todo iba bien, el programador verificaba que su rutina funcionaba correctamente, pero si había algún error (algo que con la complejidad creciente de los programas era cada vez más habitual) el programador tenía que volver a su despacho, averiguar dónde estaba el error, corregirlo y volver a testearlo, esperando que todo funcionara correctamente.

Este modelo de desarrollo de programas era extremadamente subóptimo. Resultaba evidente que los programadores que normalmente trabajaban con rutinas cortas tenían que poder ejecutar con más prioridad que los largos y costosos programas que ya estaban en fase de producción.

Una primera aproximación para solucionar esta disyuntiva fue el **sistema operativo CTSS**² (1961), desarrollado para las plataformas IBM 7090 y IBM 7094. Este sistema operativo fue el primero que permitió la interacción concurrente de diferentes usuarios con la máquina permitiendo, por ejemplo, que los programadores pudieran procesar pequeñas rutinas mientras en segundo plano se seguían ejecutando los costosos procesos de perdición o análisis de datos. Este cambio de paradigma supuso uno de los mayores hitos en la computación, ya que cambió drásticamente la forma en la que los usuarios interactuaban con la máquina, y abrió el campo a un futuro lleno de posibilidades, cuyo máximo exponente de la época fue el **sistema operativo Multics**, el primer sistema operativo de tiempo compartido.

⁽²⁾ Siglas en inglés de *compatible time-sharing system*.

Multics nació en 1969 como proyecto colaborativo entre el MIT, General Electric y Bell Lab con la ambición de transformar la computación en un servicio bajo demanda, como la electricidad o el teléfono. La idea era que los usuarios, utilizando un terminal desde sus lugares de trabajo, pudieran conectarse con una gran máquina donde, bajo demanda, ejecutarían sus trabajos. Para lograr este fin, se introdujeron numerosos avances técnicos, como las librerías dinámicas o el sistema de ficheros jerárquico. Aún con todas estas novedades, el éxito del sistema operativo fue limitado, dado que las capacidades compu-

tacionales de la época no permitían escalar el sistema a los niveles que la arquitectura requería. Aun así, la influencia que tuvo Multics en el diseño de sistemas operativos influyó a la gran mayoría de sistemas operativos modernos, desde los utilizados en microcomputadores a los utilizados *mainframes*. De hecho, más de medio siglo después de la introducción de Multics, el concepto de *computación bajo demanda* es más actual que nunca. Hoy en día, cualquiera de nosotros puede acceder a cualesquiera que sean nuestros requisitos de computación utilizando los servicios de *cloud computing*, como Amazon EC2, Google Cloud o Microsoft Azure.

1.4. Ordenadores personales

Todos los sistemas de computación que hemos visto hasta ahora estaban enfocados al análisis de grandes (para la época) volúmenes de datos o a la realización de costosas simulaciones. Su uso estaba restringido al ámbito empresarial o académico. Todo estaba a punto de cambiar en la década de 1970 con la aparición de los **circuitos integrados a gran escala** (LSI), una tecnología que permitió reducir considerablemente el tamaño de los ordenadores, así como su coste.

La aparición del microprocesador Intel 4004 y su sucesor, el 8080, en 1971 y 1974, respectivamente, abrieron el camino a la computación personal, el paradigma dominante hoy en día.

En aquella época, los grandes proveedores de ordenadores como IBM o GE estaban totalmente enfocados a la producción de grandes máquinas de cómputo. Desde su punto de vista, la aparición de los circuitos integrados supuso un aumento significativo en la capacidad de cómputo, dado que eran capaces de introducir muchos más transistores en sus máquinas. La posibilidad de utilizar microprocesadores como los Intel para producir máquinas más pequeñas de uso personal estaba fuera de sus planes. Al fin y al cabo, ¿quién querría tener un ordenador en casa? A la larga, este razonamiento supuso el final de muchas de esas empresas.

Unos pocos entusiastas vieron el potencial en el hecho de que cada usuario dispusiera de su propia máquina, sin duda mucho menos potente que los grandes *mainframes*, pero con un control absoluto sobre lo que se ejecutaba en ellas. Muchos de esos primeros usuarios vieron cómo, por un precio realmente competitivo, podían disponer en su propia casa de una tecnología solo al alcance de las grandes corporaciones, eso sí, con una capacidad de cómputo reducida.

Ejemplo: el Altair 8800

En 1974 la empresa MITS introducía el Altair 8800³ (figura 4) basado en el microprocesador Intel 8080, por un precio muy asequible, 439 dólares. La memoria de este ordenador era realmente limitada, con una versión base de 1024 bytes ampliables. El procesamiento de datos en esta máquina era muy rudimentario y no disponía de las tecnologías de

⁽³⁾Podéis acceder a un emulador de esta máquina en el siguiente enlace: www.s2js.com/altair/

multiprogramación de sus hermanos mayores, pero abrió el camino a lo que ha sido la mayor revolución tecnológica desde la revolución industrial.

En los años siguientes, cada vez más compañías vieron el tremendo potencial de los ordenadores personales y se avanzó a gran velocidad en su desarrollo. A principios de los ochenta, los ordenadores personales a través de sus sistemas operativos ya ofrecían avanzadas **técnicas de multitarea**, con las que el usuario de manera transparente podía ejecutar múltiples programas a la vez. Hoy en día es relativamente habitual tener más de 1.000 procesos corriendo a la vez. La miniaturización, a su vez, hizo la computación omnipresente en nuestras vidas: teléfonos móviles o relojes que tienen millones de veces más poder de cálculo que aquellos ordenadores primigenios.

Figura 4. Altair 8800 conectado a la máquina de teletipo Model 33 ASR



1.5. El procesamiento por lotes en la actualidad

Con todos estos avances, es fácil pensar que el antiguo paradigma del procesamiento por lotes ha quedado obsoleto. Nada más lejos de la realidad. Hoy en día todavía existen numerosos entornos en los que se utiliza este enfoque a la gestión de tareas. Los *mainframes* no han desaparecido de nuestra vida, siguen allí, cada vez más potentes, gestionando cantidades ingentes de transacciones. Los superordenadores utilizados para procesar simulaciones científicas hacen uso de las técnicas descritas para gestionar sus inmensos recursos. Como veremos en el siguiente apartado, el uso de sistemas de *batch processing* sigue vivo en muchos de los sistemas operativos que podemos encontrar en nuestros ordenadores personales. Finalmente, esta técnica es indispensable para el procesamiento de *big data*, como vais a ver a lo largo de esta asignatura.

2. Sistemas de procesamiento por lotes en entornos UNIX

2.1. Introducción

En el apartado anterior hemos visto cómo los **sistemas de procesamiento por lotes** fueron introducidos en la década de los sesenta en los grandes y costosos sistemas *mainframe*. Su misión principal era optimizar al máximo el uso de los recursos de estos sistemas, automatizando la ejecución de tareas de manera secuencial. En los sistemas que utilizaban tarjetas perforadas, esto consistía en apilarlas de manera adecuada y dejar que el ordenador se encargara de manera automática de la lectura, ejecución, registro y cambio de contexto para cada una de las tareas.

Aunque hoy en día donde vivimos en entornos multiproceso donde la mayoría de las tareas que ejecutamos se ejecutan en tiempo compartido, aún podemos encontrarnos casos donde la aplicación de técnicas de procesamiento por lotes es de extrema utilidad. Es por este motivo que los sistemas operativos modernos aún incluyen utilidades que de diferentes maneras nos permiten organizar la ejecución de tareas de una manera muy similar a cómo lo haríamos en un sistema de procesamiento puramente por lotes.

2.2. Automatización de la ejecución

En este apartado vamos a centrarnos en las herramientas que podéis encontrar en la mayoría de los sistemas derivados de UNIX, como Linux o macOS. Utilizaremos estos conceptos para introducir las herramientas para automatizar tareas *batch* `at`, `batch`, `atq`, `atrm`, `nice` y `CRON`, comparándolas y viendo para cada una de ellas diferentes casos de uso.

Estas herramientas pueden verse como una adaptación de los sistemas de procesamiento por lotes que podemos encontrar en los *mainframes*, ya que con su uso podemos regular y priorizar el flujo de ejecución.

Ved también

En el apartado 4 de este módulo encontraréis una introducción a los conceptos básicos de UNIX, así como una pequeña introducción al *scripting* en *bash*.

2.2.1. Comando *nice*

En un entorno multiproceso, donde los procesos se ejecutan de manera concurrente, resulta útil poder definir diferentes prioridades de uso de la CPU a cada proceso. Imaginemos que tenemos un proceso no prioritario encargado de copiar archivos de una carpeta a otra que se ejecuta concurrentemente con otro proceso que se encarga de analizar los paquetes enviados a través de la

red. Dado que este último proceso es crítico y que el rendimiento de la red dependerá de cómo de rápido pueda analizar cada uno de los paquetes, parece obvio que este consuma más recursos de CPU que un proceso que realiza una copia de archivos. Para regular este tipo de casos, los sistemas UNIX ponen a nuestra disposición el comando `nice`.

El funcionamiento de este comando es realmente simple. En UNIX cada proceso tiene una prioridad conocida como *nice*ness que se mueve en un rango de -20 (máxima prioridad) a 19 (mínima prioridad). El valor por defecto de un nuevo proceso es 0. Para arrancar un proceso con una prioridad diferente solo tenemos que invocar el comando `nice` seguido del parámetro -n, el valor de *nice*ness que queremos asignar y finalmente el nombre del proceso que queremos ejecutar. Si queremos ejecutar el *script* de copia de archivos (`copyfiles.sh`) con la mínima prioridad posible, solo tendríamos que ejecutar:

```
nice -n 19 sh copyfiles.sh
```

Adicionalmente, podéis comprobar el nivel de *nice*ness de cada uno de los procesos que están corriendo mediante el comando:

```
ps -eo pid,ni,comm
```

2.2.2. Comandos `at`, `atq`, `atrm` y `batch`

Determinadas tareas deben ejecutarse a una hora determinada o cuando las condiciones del sistema lo permitan. Imaginemos un proceso que tiene que hacer una migración masiva de datos de un servidor a otro o un proceso que tiene que evaluar millones de operaciones bancarias. Estos tipos de procesos consumen un alto volumen de recursos. En ocasiones nos interesaría que estos no se ejecutaran cuando hay mayor demanda del sistema sino en un momento de menor actividad, por ejemplo, fuera del horario de oficina. Los sistemas UNIX ofrecen un conjunto de herramientas que nos permiten programar **cuándo** un proceso se debe ejecutar.

El comando `at` nos permite definir ese **cuándo**. Para utilizarlo debemos introducir el comando `at` seguido de la fecha futura de ejecución del programa y, seguidamente, el comando que se desea ejecutar.

Ejemplo

Suponed que hoy es el 1 de abril de 2050, y tenemos una PEC para entregar en 5 días, el 5 de abril a las 21:00. Como somos expertos programadores en `bash` hemos creado un pequeño *script*, `entregar_practica.sh`, que comprime un determinado directorio con resultados, y envía el fichero resultante al servidor de la UOC. Como no queremos olvidar la entrega, podríamos utilizar el comando `at` para programar su ejecución cinco minutos antes de que venza el plazo:

```
at 8:55 PM April 5 2020 < entregar_practica.sh
```

Fijaros que redirigimos el contenido del *script* al comando `at` utilizando el carácter `<`. Además, como podéis observar, el formato de fecha que hemos especificado es bastante completo, hora y fecha completa. Afortunadamente, el comando `at` permite introducir la fecha en múltiples formatos. Veamos algunos ejemplos, suponiendo que la fecha actual es viernes, 1 de abril de 2050 a las diez y un minuto de la mañana.

Fecha	Equivalente
April 5	10:01 AM April 5 2020
noon	12:00 AM April 1 2020
Sunday	10:01 AM April 3 2020
noon tomorrow	10:01 AM April 2 2020
JUN	10:01 AM June 1 2020
4:30 PM	04:30 PM April 1 2020

Como podéis comprobar tenéis multitud de opciones, además incluso podéis aplicar operaciones algebraicas.

Fecha	Equivalente
now + 10 minutes	10:11 AM April 1 2020
5 PM + 1 day	17:00 AM April 2 2020
now + 5 years	10:01 AM April 1 2025

Una vez tenemos nuestra tarea programada, podemos comprobar si esta está correctamente registrada. Para ver el registro de tareas programadas solo tenéis que ejecutar el comando `atq`. Veréis que aparecen las instrucciones programadas precedidas por un número que indica la `id` del trabajo. Este identificador único resulta útil si queremos comprobar los detalles de la solicitud con el comando `at -c id o` para eliminar tareas que no queremos que se ejecuten con el comando `atrm` sucedido por el identificador.

Los comandos relacionados con el `at` son útiles cuando sabemos cuándo se tiene que ejecutar el programa, pero a veces lo que queremos es que un programa se ejecute cuando la carga permita ejecutarlo sin que este entorpezca la ejecución de procesos más prioritarios. Este escenario os recordará al que visteis en el primer apartado del módulo, cuando describíamos los sistemas multiprogramación. Los entornos UNIX disponen del comando `batch`⁴ para realizar esta tarea. Supongamos que tenemos una tarea no prioritaria `proceso_no_prioritario.sh` que puede ejecutarse cuando el sistema no está ocupado en otras tareas. Deberíamos ejecutar:

```
batch < proceso_no_prioritario.sh
```

⁽⁴⁾Podéis utilizar indistintamente el comando `at -b`. La ventaja de esta alternativa es que nos permite especificar parámetros adicionales como la cola a la que queremos asignar el proceso.

La tarea quedará programada para su ejecución cuando la carga del sistema sea inferior a 1.5. Para gestionar las tareas programadas con el comando `batch` podéis utilizar las sentencias `atq` y `atrm`, las mismas que utilizamos para el comando `at`.

Ejemplo

Suponed que tenemos una aplicación programada con `at` y otra con `batch`. El comando `atq` devolvería:

```
1  Tue Apr 13 02:02:00 2050 a user
2  Fri Apr 16 01:01:00 2050 b user
```

Fijaros que entre la fecha y el nombre de usuario aparece una letra, `a` para la primera tarea, `b` para la segunda. Esta letra representa la cola, `a` representa la prioridad más alta y `z` la menos prioritaria. La prioridad en este caso es cuantificada en *nicesness*. Podéis especificar la cola que queréis utilizar utilizando el parámetro `-q` del comando `at`. Para procesos enviados con el comando `at`, la prioridad por defecto es `a`, y para procesos enviados con el comando `batch`, esta prioridad es `b`.

2.2.3. Comando `cron`

Como hemos visto en el apartado anterior, en muchas situaciones resulta sumamente interesante poder especificar cuándo se ejecutan nuestras tareas. En algunos de estos casos estas tareas se deben ejecutar con cierta periodicidad.

Ejemplos de tareas periódicas

Imaginad por ejemplo los *scripts* encargados de realizar copias de seguridad, en este caso nos puede interesar realizar una copia de seguridad cada día a una hora determinada; o por ejemplo un programa que nos envíe un correo cada mes con el número de horas que cada usuario ha estado activo en el sistema.

Para este tipo de tareas, podemos utilizar `cron`.

Para interactuar con esta utilidad debemos hacer uso del comando `crontab`, que nos permitirá definir la tabla de aplicaciones programadas.

Ejemplo: uso de `crontab`

Supongamos que deseamos ejecutar el programa `crear_backup.sh` cada día las doce de la noche. Lo primero que tendríamos que hacer es ejecutar el comando `crontab -e` para acceder a la edición de la tabla. Este comando nos abrirá el editor de texto que tengamos definido por defecto y en el podremos introducir la periodicidad de nuestro programa. La sintaxis que se debe utilizar para este fin queda definida por el siguiente formato:

```

* * * * * <comando a ejecutar>
| | | | |
| | | | |----- minuto (0 - 59)
| | | | |----- hora (0 - 23)
| | | | |----- día del mes (1 - 31)
| | | | |----- mes (1 - 12)
| | | | |----- día de la semana (0 - 6) (De domingo a sábado)
```

De esta manera, si queremos ejecutar el comando cada día a las doce de la noche, tendríamos que introducir la siguiente línea:

```
0 0 * * * sh crear_backup.sh
```

Fijaros que el carácter `*` se utiliza como comodín y significa *cualquiera*. Este formato ofrece mucha flexibilidad, incluidas listas y rangos, por ejemplo:

Formato <code>crontab</code>	Equivalencia
<code>0 * * * *</code>	cada hora en punto
<code>0 1 * * *</code>	cada día a la una de la madrugada
<code>30 12 1 * *</code>	cada principio de mes a las 12:30
<code>10 10 * * 1</code>	cada lunes a las 10:10
<code>* * * * *</code>	cada minuto
<code>*/10 * * * *</code>	cada 10 minutos
<code>0 12 * * * 1-5</code>	de lunes a viernes a las 12:00
<code>30 8,10,12 * * *</code>	cada día a las 8:30, 10:30 y 12:30

Para editar o introducir más tareas periódicas solo tenéis que utilizar el comando `crontab -e` otra vez. Si por el contrario solo queréis revisar la lista de trabajos programados utilizad el comando `crontab -l`. Finalmente, si queréis desactivar todas las tareas solo tenéis que utilizar el comando `crontab -r`.

3. Clústeres de computación: sistemas de colas

Hasta ahora hemos visto cómo los avances en computación han ido de la mano con los crecientes requisitos de los usuarios y han modificado de manera trascendental la forma en la que, a día de hoy, procesamos la información. También hemos visto cómo los sistemas con los que cada uno de nosotros trabajamos a diario tienen sistemas para gestionar de manera sistemática cómo y cuándo se ejecutan determinados programas, de manera similar a cómo lo gestionaríamos en un sistema de procesamiento por lotes clásico. En este apartado vamos a centrarnos en explicar cómo las herramientas de procesamiento por lotes son especialmente importantes en los sistemas de computación de alto rendimiento formadas por cientos de ordenadores, donde los recursos, extremadamente cotizados, tienen que ser minuciosamente gestionados.

La idea de conectar varios ordenadores entre ellos para repartir la computación de las tareas más complicadas no es nueva. Los conceptos teóricos se establecieron a mediados de la década de 1960. Para finales de esa década ya habían puesto los primeros prototipos en funcionamiento, y para finales de los setenta ya estaban presentes en la mayoría de grandes proyectos de computación: predicción de modelos meteorológicos, grandes simulaciones científicas, proyecciones militares, etc. Su expansión se debía a que, a diferencia de los supercomputadores monolíticos, o clústeres de computación, son mucho más escalables y económicos. Hoy en día, si miramos los *rankings* de los sistemas de cómputo más potentes, sin duda encontraremos representantes de esta familia de máquinas en las primeras posiciones.

En este apartado vamos a focalizarnos en:

- Entender cuáles son las características básicas que definen la arquitectura de los clústeres de computación.
- Ver cómo interactúan los usuarios con ellos mediante un sistema de colas que permite priorizar los trabajos de acuerdo con un sistema de reglas preestablecido.
- Comparar las diferencias entre los clústeres de computación y los sistemas que actualmente se utilizan en el análisis de datos *big data*.

3.1. Arquitectura

De manera genérica, un clúster se puede definir como una agrupación de ordenadores, llamados nodos, conectados entre sí y capaces de realizar costosas operaciones de cómputo en paralelo.

El número de nodos que componen un clúster puede variar, desde unos pocos hasta centenares, dependiendo de los requisitos de sus propietarios. Cada uno de los nodos puede tener especificaciones diferentes: por ejemplo, podemos encontrarnos con nodos con GPU destinados a labores de inteligencia artificial, ordenadores con grandes cantidades de memoria RAM destinados a tareas bioinformáticas o nodos con una gran cantidad de núcleos de cómputo. Toda esta gran variedad de recursos está gestionada por una máquina central, el *scheduler*, que juega un papel fundamental en el sistema, ya que es la encargada de priorizar, asignar recursos y gestionar la ejecución de las tareas que los usuarios solicitan.

Ved también

En el apartado 3.2. veremos cómo funciona el sistema de colas donde los usuarios envían sus programas.

Otra característica importante de los clústeres es la **gestión de los datos**. En la gran mayoría de configuraciones, los datos que cada usuario va a utilizar están almacenados en máquinas específicas que, de manera centralizada, permiten a cada uno de los nodos del sistema crear y modificar los datos allí guardados y acceder a ellos. Este sistema es esencial, ya que no sabemos de antemano a qué nodos se van asignar cada uno de los trabajos y, por tanto, debemos asegurarnos de que tanto la entrada como la salida comparten un espacio común. Además, dadas las características de la programación en paralelo, algunas de las tareas necesitarán comunicarse con otros procesos o utilizar los datos que estos han generado previamente. Por lo tanto, es importante que el acceso a los datos sea compartido por todos los nodos y que sea muy rápido. Uno de los elementos que caracterizan a los clústeres es, precisamente, la **comunicación entre procesos**.

Para repartir el cómputo de una determinada tarea entre varios ordenadores existen diferentes métodos. Uno de los métodos más simples es dividir la tarea de tal manera que cada uno de los procesos que se ejecuten en cada máquina sean independientes.

Ejemplo: procesos independientes

Las simulaciones científicas que tienen un comportamiento estocástico, como las que se utilizan para estudiar la expansión del universo, requieren repetidas ejecuciones para generar suficiente significancia estadística para poder evaluar hipótesis. Estas simulaciones pueden ejecutarse independientemente, y solo es necesario agregar los resultados una vez hayan finalizado. Este es el caso más claro de paralelismo.

Otros procesos, en cambio, necesitan comunicarse entre ellos para cumplir sus objetivos.

Ejemplo: procesos que se comunican

Supongamos que queremos calcular el resultado de multiplicar un vector por una matriz de un tamaño mayor al que cabría en la memoria del ordenador. En este caso, podemos distribuir la matriz entre varios nodos. Cada uno de los nodos se comunicará con los otros para solucionar el problema.

La comunicación entre procesos se realiza típicamente a través de la interfaz MPI (*message passing interface*). Aunque imprescindible, en este tipo de paralelismos el programador debe intentar minimizar la comunicación entre diferentes procesos, ya que esta tiene un impacto significativo en el tiempo de cómputo.

Sea cual sea el tipo de paralelización que vaya a utilizarse, los recursos de los clústeres son muchos pero limitados y, además, se comparten con otros usuarios. Para gestionar eficientemente los recursos del sistema se utiliza un **sistema de colas** que sigue los principios del procesamiento por lotes introducidos en la década de los sesenta. En el próximo apartado introduciremos los elementos básicos de sistema de colas de los clústeres.

3.2. Sistema de colas

En el primer apartado de este módulo hemos visto cómo en los primeros sistemas de cómputo se utilizaban pilas de tarjetas perforadas para organizar los trabajos de los programadores. Estas se apilaban y se cargaban en el sistema de manera secuencial, estos son los primeros sistemas de procesamiento por lotes. Más tarde, con la introducción de la multiprogramación se optimizó el uso de cada uno de los componentes. Los *schedulers*, las máquinas que gestionan la ejecución y los recursos en los clústeres, están muy inspirados en estos sistemas, aunque han incrementado mucho su eficiencia y añaden una enorme flexibilidad a la hora de priorizar el orden en que se ejecutan los procesos. Los elementos básicos para manejar todo este sistema son las colas de procesos.

En una configuración básica del *scheduler*, este funciona como funcionaría una pila de tarjetas perforadas. Los programas son añadidos a la cola por orden de llegada y se van ejecutando por orden a medida que el clúster tiene recursos disponibles. Este tipo de procesamiento se conoce como *FIFO*, por sus siglas en inglés (*first in first out*).

Esta configuración resulta muy simple, pero en muchas ocasiones es necesario ejecutar algunos programas antes que otros. Imaginad un clúster donde algunas de las tareas que se ejecutan están dedicadas a calcular la estructura de las proteínas. Como este tipo de problema requiere muchísima capacidad de cómputo, mientras estas se ejecutan todas las que se vayan añadiendo a la cola tendrán que esperar un tiempo considerable antes de ser ejecutadas. Es precisamente por este motivo que se introduce el concepto de *cola prioritaria*, donde la posición en esta queda determinada por una serie de premisas establecidas por el administrador del sistema.

En la práctica existen numerosos programas capaces de especificar y administrar colas de procesos. En este apartado utilizaremos *slurm* para ilustrar cómo funcionan estos sistemas. Veamos cómo se puede enviar un programa a ejecutar en un clúster.

Ejemplo: enviar un programa a ejecutar en un clúster

Imaginemos un programa muy simple, `simple.sh`, como el siguiente:

```
#!/bin/bash
hostname
```

Este programa lo único que hace es mostrar por la salida estándar el nombre de la máquina donde se ha ejecutado. Para enviarlo a la cola de procesos, simplemente tenemos que conectarnos a la máquina donde se encuentra el *scheduler* y ejecutar:

```
sbatch simple.sh
```

Aunque funcional, este tipo de ejecución introduce un problema al planificador, ya que, si no lo especificamos, no puede saber cuántos recursos va a necesitar nuestro programa (en este caso muy pocos). Cuando se envían programas al clúster es normal especificar qué recursos estimamos que va a necesitar nuestro programa para su ejecución. Estos parámetros se pueden especificar tanto dentro del propio *script* como desde la llamada a *sbatch*. Veamos cómo podemos modificar el anterior *bash script* para introducir esta información.

```
#!/bin/bash
#SBATCH -c 1                # Número de cpus requerido
#SBATCH -t 0-00:05         # Tiempo estimado de ejecución en el formato
                           # D-HH-MM
#SBATCH --mem=100           # Memoria que requiere la ejecución en MB
#SBATCH -o hostname_%j.out # Fichero donde la salida standard se
                           # redirigirá,
                           # el fichero incluirea el id del trabajo (%j)
hostname
```

Entre otros factores, este tipo de información es la que el planificador utilizará para asignarnos una prioridad de ejecución. *Slurm* permite definir de manera muy flexible qué factores se utilizarán para ejecutarse. Los más importantes son:

- **La edad del proceso (`age_factor`):** procesos que hayan estado más tiempo en la cola tendrán más prioridad.
- **Los recursos asignados (`job_size`):** dependiendo de las características del clúster los administradores pueden escoger asignar prioridades más altas a los procesos que requieran más recursos o menos.
- **Ajuste a los recursos requeridos (`fair-share_factor`):** como hemos visto, el usuario es el que especifica los recursos que va a utilizar un trabajo. En determinados sistemas se priorizan los trabajos que más recursos han solicitado. En estos casos, el usuario podría estar tentado a solicitar más recursos de los que realmente necesita. Para evitar este tipo de situaciones,

los sistemas de colas pueden penalizar los trabajos que no utilizan todos los recursos especificados.

- **Partición a la que se asigna el trabajo (`partition_factor`):** como habéis leído, los clústeres pueden estar formados por una gran variedad de máquinas, cada una de ellas con diferentes características técnicas. Con las particiones, el administrador puede agrupar las máquinas de la manera que más le convenga, por ejemplo, podemos tener una partición compuesta por las máquinas con GPU, otra con ordenadores con grandes cantidades de memoria, etc. Una misma máquina puede estar asignada a más de una partición. Para graduar qué partición tiene preferencia para utilizar las máquinas compartidas para sus trabajos, podemos asignar una prioridad a cada partición.
- **Calidad del servicio (QOS):** una manera muy sencilla de gestionar un clúster es predefiniendo una serie de requisitos prototipo. Por ejemplo, podemos definir *short*, para los procesos que utilizan menos de dos CPU y 16 GB de RAM y requieren menos de dos horas para completarse. De la misma manera, podemos definir *medium* para los procesos de menos de tres CPU y 64 GB y que se ejecuten en menos de dos días. En este caso, podríamos asignar una prioridad más alta a los programas asignados a *short*, media a los asignados a *medium* y baja para todos los demás.
- **Niceness:** igual que hemos visto en los sistemas UNIX, podemos asignar un valor de *niceness* a nuestros programas.

Todos estos factores se pueden combinar y ponderar de la manera que consideremos adecuada, utilizando pesos. Una vez definidos, *slurm* calculará la prioridad de cada proceso utilizando la siguiente fórmula y los añadirá a la cola de ejecución.

```
Job_priority = (PriorityWeightAge) * (age_factor) +  
               (PriorityWeightJobSize) * (job_size_factor) +  
               (PriorityWeightFairshare) * (fair-share_factor) +  
               (PriorityWeightQOS) * (QOS_factor) +  
               (PriorityWeightPartition) * (partition_factor) +  
               - nice_factor +  
               etc.
```

3.3. Diferencias con los sistemas *big data*

Aunque pueda parecer que los sistemas aquí relatados para las arquitecturas HPC puedan aplicarse a la mayoría de los problemas que se tratarán dentro del paradigma del *big data* para el procesamiento de lotes, existen diferencias

significativas entre estas técnicas y las que aprenderemos en esta asignatura. Estas diferencias van desde la manera que tienen los usuarios de interactuar con el sistema, hasta la arquitectura de hardware utilizada.

La gran mayoría de estas diferencias radican en los objetivos que se intentaban conseguir cuando se diseñaron estos dos sistemas. Como hemos visto, la aparición de los primeros clústeres data de la década de los sesenta, y desde ese momento su finalidad ha sido la de expandir la capacidad de cómputo en paralelo para atacar problemas extremadamente complejos como los que se afrontan en simulaciones científicas. En cambio, los sistemas que vais a ver en esta asignatura son de creación mucho más reciente, Google introduce el modelo de programación MapReduce a principios de la década de los 2000, y nacen en pleno auge de la era de la información. Su objetivo es tratar de la manera más eficiente posible la cantidad ingente de datos que este nuevo mundo produce cada día. Dado que el foco principal de los sistemas *big data* es el análisis de grandes volúmenes de datos (de aquí el nombre), sus ventajas e inconvenientes están caracterizadas por este hecho.

Los sistemas *big data* son altamente **escalables**: están compuestos desde unos pocos ordenadores a miles de ellos; de hecho, pueden añadirse ordenadores al sistema de manera muy sencilla. En cambio, escalar los clústeres no resulta tan fácil, por diversos motivos. El primero de ellos es la infraestructura. Los clústeres necesitan conexiones de altísima velocidad para mantener la comunicación entre procesos, lo que hace que en la práctica estos ordenadores tengan que estar bajo una misma infraestructura. Otro motivo por el que los requisitos de infraestructura son más estrictos en los clústeres de computación es el **sistema de ficheros**. Como habéis visto, en los clústeres los ficheros están centralizados y todos los nodos tienen acceso a todos ellos. En los sistemas *big data* esto es totalmente diferente, los archivos acostumbran a estar distribuidos entre todos los nodos del sistema y cada nodo procesa solo su parte. Esto supone una mayor escalabilidad del sistema con el volumen de datos y una mayor tolerancia a los fallos.

Estas características suponen ventajas e inconvenientes que iremos viendo a lo largo del curso.

Los sistemas que estudiaremos están enfocados al análisis masivo de datos. La programación de aplicaciones destinadas a este fin es muy eficiente y se puede producir software mucho más rápido de lo que se podría si tuviéramos que paralelizarlo utilizando, por ejemplo, MPI. En cambio, resulta un factor limitante cuando deseamos utilizar esta misma infraestructura para aplicaciones de un hábito más general, en estos casos los sistemas de clústeres de computación son más flexibles.

4. Introducción a los sistemas UNIX

El sistema operativo UNIX fue introducido en la década de los setenta como paradigma del sistema operativo multitarea y multiusuario. A día de hoy, los sistemas derivados/similares⁵ al sistema desarrollado en AT&T por Dennis Ritchie y Ken Thompson son omnipresentes en la mayoría de dispositivos electrónicos.

⁽⁵⁾Diferenciamos entre sistemas puramente UNIX, que son los que se derivan de la licencia original de AT&T, como son BSD, Solaris o macOS, y aquellos que son similares, o *UNIX-like*, que principalmente están formados por los sistemas Linux y sus variantes.

Desde los superordenadores que se utilizan para simular complejos sistemas físicos hasta los más elementales electrodomésticos o aspiradoras. De hecho, la gran mayoría de nosotros llevamos un sistema UNIX corriendo en nuestro bolsillo ya que tanto el núcleo de Android como iOS, los dos principales sistemas operativos para móviles, corresponde a esta categoría de sistemas operativos.

Este tipo de sistemas puede dividirse en tres grandes partes:

1) **El núcleo o *kernel*:** en la parte más profunda del sistema operativo encontramos el *kernel*. Su misión principal es hacer de interfaz entre el software y el hardware, asegurar la integridad de las acciones que se ejecutan y gestionar y asignar los recursos del sistema.

2) **La *shell*:** es una interfaz de comunicación entre el usuario y el sistema operativo. La *shell* se utiliza para transformar tareas específicas en alto nivel en acciones que el ordenador pueda ejecutar. Estas tareas pueden estar definidas en programas aumentando así el nivel de abstracción. Existen dos tipos principales de *shell*:

- **La línea de comandos:** utilizaremos este tipo de *shell* cuando queramos interactuar con el sistema operativo de manera textual.
- **La interfaz gráfica:** con la introducción a principios de los ochenta de las interfaces gráficas, la manera de interactuar con el *kernel* cambió de manera radical. En este tipo de *shell*, la interacción con el *kernel* se realiza utilizando metáforas visuales, como arrastrar un fichero de una carpeta a otra o eliminar un archivo tirándolo a la papelera.

3) **Los programas:** a más alto nivel encontramos los programas, un conjunto específico de instrucciones que realizan una tarea concreta.

En este apartado utilizaremos la línea de comandos *bash*⁶ para ilustrar la interacción con el sistema operativo. Hemos escogido esta opción ya que, aparte de ser libre, es la línea de comandos por defecto en los sistemas UNIX más populares. En el resto del apartado detallaremos dos aspectos fundamentales de *bash*:

⁽⁶⁾ Acrónimo en inglés para *bourne-again shell*, ya que originariamente se desarrolló para el proyecto GNU como sustituto de su versión licenciada en UNIX, la *bourne shell*.

- La interacción con el sistema de ficheros
- Una breve introducción al *bash scripting*

4.1. Sistema de ficheros

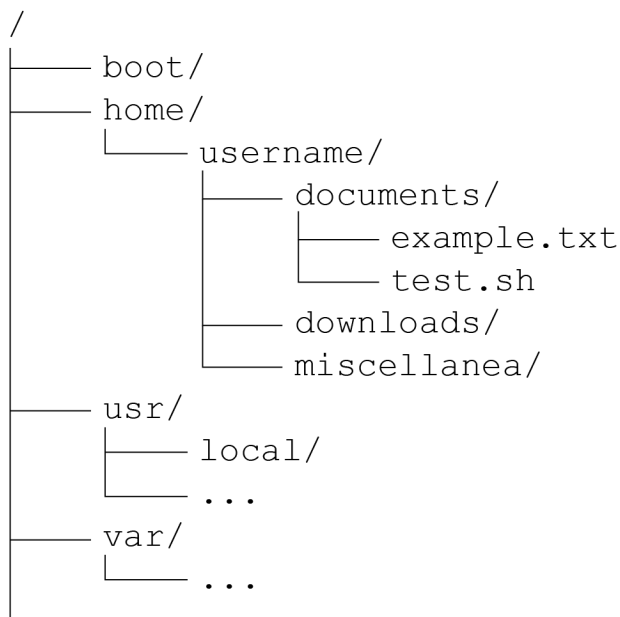
4.1.1. Estructura

Muchas de las acciones que ejecutamos cada día en nuestros ordenadores están relacionadas con la **gestión de ficheros**. Crear, borrar, mover o copiar estos elementos se ha convertido en una tarea cotidiana.

Los sistemas UNIX utilizan una estructura anidada de directorios para organizar los archivos. La raíz jerárquica de este sistema de directorios se representa con el símbolo de la barra (/). De esta cuelgan todos los directorios.

Cada uno de estos directorios a su vez contiene directorios anidados, como podemos ver en la figura 5.

Figura 5. Ejemplo de estructura de directorios en un entorno UNIX



La localización de un archivo queda definida por su ruta. Así, la localización del fichero `ejemplo.txt` en el sistema descrito en la figura 5 queda definida por la ruta `/home/username/documents/ejemplo.txt`. Esta ruta se denomina «absoluta» ya que contiene el camino completo desde el directorio raíz `/`. Los archivos también se pueden referenciar de manera relativa a la posición actual. Así, si nos encontramos en la carpeta `username`, podemos referenciar al fichero utilizando la ruta relativa `./documents/ejemplo.txt`. Las rutas relativas más comunes son:

- `.`: referencia al directorio donde nos encontramos actualmente.
- `..`: referencia al directorio superior en la jerarquía. Si nos encontramos en el fichero `username`, la ruta `../` hace referencia al directorio `home`.
- `~`: referencia al directorio principal del usuario. En la mayoría de sistemas UNIX cada usuario tiene definida una carpeta personal. En los sistemas Linux, esta normalmente cuelga del directorio `/home`. Por defecto, muchas de las carpetas que utilizaremos en nuestro día a día van a colgar de nuestra carpeta personal, por lo que esta ruta relativa resulta sumamente útil para interactuar con la *shell*.

Minúsculas y mayúsculas

Es importante recordar que en algunos sistemas (mayoritariamente Linux) la ruta es sensible a letras minúsculas o mayúsculas. Así, dos ficheros en el mismo directorio con nombre `Ejemplo.txt` y `ejemplo.txt` representan entidades diferentes.

4.1.2. Interacción

En este apartado vamos a describir a grandes rasgos cómo interaccionar con este sistema de ficheros utilizando *bash*.

Por regla general, una vez accedemos a la *shell* nos encontramos en el directorio de nuestro usuario (`~/`). Aun así, podemos comprobar en cualquier momento nuestra posición actual dentro de la jerarquía de directorios utilizando el comando `pwd`. Si ejecutamos el comando en nuestro directorio personal veremos la ruta absoluta de este dentro del sistema de archivos.

```
pwd
/home/username
```

Donde `username` corresponde a vuestro nombre de usuario.

Una vez nos hemos localizado dentro del árbol de directorios, podemos empezar a desplazarnos por él. Para cambiar de directorio utilizaremos el comando `cd` seguido de la ruta (absoluta o relativa) de destino. Así, para acceder al directorio `documents` desde la carpeta de usuario podemos hacerlo con la ruta absoluta:

```
cd /home/username/documents
```

o con la ruta relativa:

```
cd documents
```

Fijaros que para acceder a una carpeta contenida dentro del directorio en el que nos encontramos podemos omitir el atajo `./`.

Para navegar hacia arriba en la estructura jerárquica, solo podemos ejecutar:

```
cd..
```

En muchas ocasiones, no conoceremos/recordaremos de antemano la estructura completa de los directorios. Con el comando `ls` obtendremos una lista del contenido, los directorios y los ficheros de un determinado directorio.

Ejemplo

Si nos encontramos en el directorio `documents` y ejecutamos el comando `ls`, obtenemos:

```
ls
example.txt test.sh
```

Si queremos el contenido de una carpeta diferente a la actual, añadiremos su ruta (absoluta o relativa) al final del comando. Para obtener una lista del contenido de la carpeta jerárquicamente superior ejecutaremos:

```
ls ../  
documents downloads
```

La mayoría de comandos de *bash* pueden enriquecerse a través de parámetros.

Ejemplo

Añadiendo el parámetro `-l` podemos obtener detalles avanzados del contenido del directorio, como el propietario del fichero, fecha de última modificación, tamaño, permisos de acceso, etc.

```
ls -l  
-rw-r--r--. 1 username username 2475 Mar  3 12:31 example.txt  
-rw-r--r--. 1 username username 1324 Jul  5 14:07 test.sh
```

Este es solo uno de los múltiples parámetros disponibles para el comando `ls`, el resto están especificados en el manual del comando. *bash*, que ofrece una manera de consultar este manual directamente desde la línea de comandos. Si queremos acceder al manual de un determinado comando solo tenemos que utilizar la sentencia `man nombredelcomando`. Si ejecutamos `man ls`, veremos

una explicación detallada del comando `ls` y todos los parámetros que este tiene disponibles. Una vez finalizada la consulta, podemos salir del manual utilizando la tecla `q`.

Antes de adentrarnos en la modificación de archivos y directorios, debemos mencionar la existencia de un símbolo muy útil a la hora interactuar con la *shell*, el comodín. El comodín (o *wildcard*, en inglés), representado por el símbolo `*`, se utiliza para representar expresiones regulares, y significa literalmente *cualquier cosa*.

Ejemplos

La expresión regular `*.txt` significa «cualquier fichero/directorio cuyo nombre termine con la extensión `.txt`».

La expresión `t*.sh` significa «cualquier fichero/directorio cuyo nombre empiece por `t` y termine con la extensión `.sh`».

El comodín es una herramienta muy versátil que puede integrarse con los comandos habituales de `bash`.

Ejemplo

Si queremos los detalles de todos los ficheros de un directorio que terminen con la extensión `.txt` podemos utilizar el comando:

```
ls -l *.txt
-rw-r--r--. 1 username username 2475 Mar  3 12:31 example.txt
```

Una vez presentados los comandos para navegar a través de sistema de archivos vamos a presentar las herramientas básicas para manipularlo. La primera tarea que vamos a tratar es la **creación de archivos**. Podemos crear un archivo vacío utilizando el comando `touch` seguido de la ruta y el nombre del nuevo archivo.

Ejemplo

Si nos encontramos en el directorio `documents` y queremos crear un documento en el mismo directorio⁷ con el nombre `readme.txt` solo tenemos que ejecutar:

```
touch readme.txt
```

⁽⁷⁾ Si el archivo que queremos crear se encuentra en el mismo directorio donde nos encontramos, podemos no incluir la ruta.

Si en lugar de crear un archivo lo que queremos es **crear un directorio**, utilizaremos el comando `mkdir`, seguido de la ruta y el nombre del nuevo directorio. Si quisiéramos crear un directorio *scripts* en nuestra carpeta de usuario utilizaríamos el comando:

```
mkdir ~/scripts
```

Ahora que hemos creado una carpeta *script*, puede que nos interese mover todos los archivos con la extensión⁸ *.sh* guardados en la carpeta *documents*. Para **mover ficheros**, utilizaremos el comando *mv*, seguido de la ruta de origen y la ruta de destino. En este caso, deberíamos ejecutar:

```
mv ~/documents/*.sh ~/scripts
```

Fijaos que en este caso hemos hecho uso del comodín para copiar cualquier fichero cuya extensión termine en *.sh*. Además, hemos hecho uso de rutas relativas al directorio del usuario utilizando el símbolo *~*.

De manera muy similar, podemos **copiar ficheros**, simplemente en lugar de usar el comando *mv*, utilizaremos el comando *cp*, seguido de la ruta de origen y la de destino.

Ejemplo

De esta manera, si queremos copiar el fichero *readme.txt*, que acabamos de crear en la carpeta *documents*, en la carpeta *scripts*, utilizaremos el siguiente comando:

```
cp ~/documents/readme.txt ~/scripts
```

Para **eliminar archivos y directorios**⁹ utilizaremos el comando *rm*.

Ejemplo

Si quisiéramos eliminar el fichero *example.txt* del directorio *documents*, solo necesitaríamos ejecutar la siguiente instrucción:

```
rm ~/documents/example.txt
```

Estos de aquí son solo algunos de los comandos más habituales a la hora de interactuar con la línea de comandos. Hay muchos más, como:

- el comando *cat* para ver el contenido de un archivo
- *head* para mostrar solo las primeras líneas
- *chmod* para cambiar los permisos de directorios y archivos, etc.

4.2. Breve introducción al *bash scripting*

Hasta ahora hemos visto cómo, desde la línea de comandos, podemos interactuar con el sistema operativo para navegar y modificar el sistema de archivos. Si las acciones que queremos ejecutar son únicas (solo se ejecutarán una vez) o consisten en pocos comandos, interactuar directamente con la línea de comandos es suficiente. Sin embargo, en otras ocasiones nos encontraremos

⁽⁸⁾ Los *scripts* de *bash* tradicionalmente utilizan la extensión *.sh*.

⁽⁹⁾ Para eliminar directorios añadiremos al comando *rm* los parámetros *-rf* para eliminar el directorio y todo su contenido.

Ved también

En la bibliografía de este módulo podéis encontrar enlaces a material adicional que os permitirá ampliar vuestros conocimientos sobre cómo interactuar con *bash*.

⁽¹⁰⁾ En ocasiones hay comandos que para producir el efecto deseado requieren especificar de manera detallada un número elevado de parámetros.

delante de tareas repetitivas que consisten en un número de pasos no negligibles o sentencias que por su complejidad¹⁰ no son triviales. En este tipo de situaciones aprovecharemos las capacidades de *scripting* de *bash*.

Un *script* no es más que una secuencia ordenada de acciones como cualquier otro programa informático. En los *scripts* de *bash*, estas acciones pueden ser cualquiera de los comandos que utilizaríamos en la línea de comandos.

Consideremos el siguiente ejemplo:

Ejemplo

Supongamos que tenemos una carpeta `downloads` en nuestra carpeta de usuario. En este directorio se guardan automáticamente todos los ficheros que nos descargamos de internet. Cada cierto tiempo nos interesa clasificar los documentos que tenemos en esta carpeta. Concretamente queremos: guardar los ficheros con extensión `.txt` en una carpeta `text` guardada dentro de la carpeta `documents`; guardar los ficheros `.sh` dentro de una carpeta `scripts`; los `.doc` en una carpeta `word`; y finalmente los archivos con extensión `pdf` dentro de una carpeta `pdfs` también guardada en la carpeta del directorio personal `documents`.

Para cumplir este objetivo utilizando la línea de comandos, deberíamos ejecutar ocho comandos: cuatro para asegurarnos que las carpetas de destino existen y sino crearlas con el comando¹¹ `mkdir`, y cuatro más para mover los ficheros de cada tipo a sus respectivos directorios. Podemos simplificar mucho más esta actividad utilizando *scripts*.

Para ello simplemente crearemos un archivo con extensión `.sh` que contenga el siguiente código:

```
#!/bin/bash
mkdir -p ~/documents/text
mkdir -p ~/documents/scripts
mkdir -p ~/documents/word
mkdir -p ~/documents/pdfs

mv ~/downloads/*.txt ~/documents/text
mv ~/downloads/*.sh ~/documents/scripts
mv ~/downloads/*.doc ~/documents/word
mv ~/downloads/*.pdf ~/documents/pdfs
```

La primera línea del archivo, `#!/bin/bash`, contiene el *shebang* y se utiliza para especificar qué programa debe interpretar el *script*, en este caso *bash*. Acto seguido, introducimos la secuencia de comandos que nuestra acción requiere. Para ejecutar este *script* solo tenemos que utilizar el comando `sh` seguido de la ruta al archivo. Si el archivo fuese `filterFiles.sh` entonces solo tendríamos que ejecutar `sh filterFiles.sh` cada vez que quisiéramos clasificar los documentos de la carpeta `downloads`.

Este pequeño *script* es solo un ejemplo del potencial que ofrece *bash*, ya que, además de las funciones preestablecidas, podemos utilizar muchas de las funcionalidades clásicas de la programación: variables, estructuras de control de flujo, bucles, etc.

Ejemplo

Suponed que queremos aplicar la misma acción de antes a cualquier carpeta, no solo a la carpeta `downloads`. Una manera sencilla de hacerlo sería que nuestro *script* utilizara

(11) Añadimos el parámetro `-p` para asegurarnos que posibles directorios intermedios en la ruta se creen si no existen. Adicionalmente, utilizamos este parámetro para evitar que el sistema nos reporte un error cuando intentamos crear un directorio que ya existe.

Ved también

En la bibliografía encontraréis enlaces a materiales de soporte donde se explican en profundidad todas las posibilidades del *bash* a la hora de implementar *scripts*.

argumentos, y el primero de ellos contuviera la ruta a la carpeta que contiene los archivos que queremos clasificar. Para conseguirlo solo tendríamos que modificar el *script* de la siguiente manera:

```
#!/bin/bash
cd $1

mkdir -p ~/documents/text
mkdir -p ~/documents/scripts
mkdir -p ~/documents/word
mkdir -p ~/documents/pdfs

mv /*.txt ~/documents/text
mv /*.sh ~/documents/scripts
mv /*.doc ~/documents/word
mv /*.pdf ~/documents/pdfs
```

Fijaros que el cambio es muy sutil, pero incrementa enormemente la reusabilidad del *script*. En este caso \$1 hace referencia a la variable¹² que contiene el valor del primer argumento de la llamada, en este caso la carpeta desde la que queremos mover los archivos. El primer paso del *script* es movernos a ese directorio utilizando el comando `cd`. Acto seguido repetiríamos los comandos anteriores solo cambiando la ruta relativa de los ficheros que nos disponemos a mover. Así, si quisiéramos clasificar los archivos contenidos en la carpeta `miscellania` solo tendríamos que ejecutar `sh filterFiles.sh ~/miscellania`.

(12) El símbolo del dólar \$ se utiliza en *bash* para hacer referencia a que estamos tratando con el nombre de una variable.

Bibliografía

Bioinformatics. «Sun Grid Engine for beginners». <https://bioinformatics.mdc-berlin.de/intro2UnixandSGE/sun_grid_engine_for_beginners/README.html>.

Bioinformatics. «Unix for beginners». <https://bioinformatics.mdc-berlin.de/intro2UnixandSGE/unix_for_beginners/README.html>.

Both, D. (2020). «How to use cron in Linux». *Opensource.com*. <<https://opensource.com/article/17/11/how-use-cron-linux>>.

Broberg, M. (2019). «The birth of the Bash Shell». *Opensource.com*. <<https://opensource.com/19/9/command-line-heroes-bash>>.

Case Western Reserve University (2020). «GNU Bash Reference Manual». <<https://tiswww.case.edu/php/chet/bash/bashref.html>>.

Command Line Heroes. «Mainframes: The GE 225 and the Birth of BASIC» (podcast). *Red Hat blog*. <<https://www.redhat.com/en/command-line-heroes/season-4/mainframes>>.

Command Line Heroes. «Learning de BASICS» (podcast). *Red Hat blog*. <<https://www.redhat.com/en/command-line-heroes/season-3/learning-the-basics>>.

Command Line Heroes. «Personal Computers: The Altair 8800 and the Dawn of a Revolution» (podcast). *Red Hat blog*. <<https://www.redhat.com/en/command-line-heroes/season-4/personal-computers>>.

Command Line Heroes. «The Data Explosion» (podcast). *Red Hat blog*. <<https://www.redhat.com/en/command-line-heroes/season-2/the-data-explosion>>.

Command Line Heroes. «Heroes in a Bash Shell» (podcast). *Red Hat blog*. <<https://www.redhat.com/en/command-line-heroes/season-3/heroes-in-a-bash-shell>>.

Computer History Museum. «Timeline of Computer History». <<https://www.computerhistory.org/timeline/computers/>>

Computer Hope (2020). «Linux at, batch, atq, and atrm commands». <<https://www.computerhope.com/unix/uat.htm>>.

Computer Hope (2019). «Linux nice command». <<https://www.computerhope.com/unix/unice.htm>>.

Computer Hope (2019). «Linux crontab command». <<https://www.computerhope.com/unix/ucrontab.htm>>.

Copeland, J. (2006). «The Modern History of Computers». *Stanford Encyclopedia of Philosophy*. <<https://plato.stanford.edu/entries/computing-history/>>.

Gite, V. (2019). «How to Add Jobs To cron Under Linux or UNIX». *Cyberciti.biz*. <<https://www.cyberciti.biz/faq/how-do-i-add-jobs-to-cron-under-linux-or-unix-oses/>>.

Griffith, E. (2020). «What is Cloud Computing?». *PCmag*. <<https://www.pcmag.com/news/what-is-cloud-computing>>.

IBM. «What is a mainframe? It's a style of computing». IBM Knowledge Center. <https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zmainframe/zconc_whatismainframe.htm>.

IBM. «What is batch processing». IBM Knowledge Center. <https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconc_whatisbatch.htm>.

Ingram, D. (2009). *Design - Build - Run: Applied Practices and Principles for Production-Ready Software Development*. Wrox. <www.oreilly.com/library/view/design-build/9780470257630/>

Lubar, S. (1991). «A cultural history of the punch card». *Wayback Machine*. <<https://web.archive.org/web/20060830162506/http://ccat.sas.upenn.edu/slubar/fsm.html>>.

McCarthy, J. (1983). «Reminiscences on the Theory of Time-Sharing». *Profesor John McCarthy*. <<http://jmc.stanford.edu/computing-science/timesharing.html>>.

Raamana, P. R. (2018). «High performance computing tutorial». <<https://www.slideshare.net/raamana/high-performance-computing-with-checklist-and-tips-optimal-cluster-usage>>.

Scientific Programming School (2017). «Introduction to High Performance Computing». <<https://learn.scientificprogramming.io/introduction-to-high-performance-computing-hpc-clusters-9189e9daba5a>>.

StarCluster. «Sun Grid Engine (SGE) QuickStart». <<http://star.mit.edu/cluster/docs/latest/guides/sge.html>>.

StarCluster. «StarCluster - MIT». <<http://star.mit.edu/cluster/>>.

Tanenbaum, A. S.; Bos, H. J. (2015). *Modern operating systems* (4.^a ed.). Pearson Higher Education. <<http://lib.bvu.edu.vn/bitstream/TVDHBRVT/19439/1/Modern-Operating-systems.pdf>>.

Top 500 The list (2019). «Top 500 supercomputers». *Top500.org*. <<https://www.top500.org/lists/top500/2019/06/>>.

Turk, R. (2018). «(Linux and) The Enduring Magic of Unix». *Red Hat blog*. <<https://www.redhat.com/en/blog/linux-and-enduring-magic-unix>>.

Zappedia. «High Performance Computing Cluster: Types and Examples». <<https://zappedia.com/high-performance-computing-cluster/>>.

40 years of Multics, 1969 - 2009