

TEMA 1: INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

1. ¿QUÉ ES UN SISTEMA OPERATIVO?

Un **sistema operativo** es un **conjunto de programas en ejecución** llamados **procesos** que **administran el hardware de una computadora** siendo o demás **programas de aplicación o de sistema**.

Un **proceso** es un **programa cargado en memoria principal**, el SO debe estar **cargado en esta y disponer de CPU para poderse ejecutar como otro proceso**. Hay dos **tipos** de **programas**:

- **Programas de sistema:** Controlar las operaciones propias de la computadora como el SO.
- **Programas de aplicación:** Resuelven problemas específicos a los usuarios como el procesador de textos.

Las funciones y objetivos del sistema operativo son:

- Gestionar y planificar el uso, por parte de los procesos activos, de los recursos hardware y servicios de sistema para que el usuario no se preocupe.
- Proporcionar una interfaz intermediaria entre el hardware y el usuario.

1.1 EL SISTEMA OPERATIVO COMO GESTOR-ASIGNADOR DE RECURSOS

Al enfrentarse a conflictivas solicitudes de recursos, el SO debe decidir cómo asignar esto a procesos. Cualquier planificación y asignación de recursos debe tener en cuenta tres factores:

- **Equitatividad:** Ante procesos similares, es deseable que a todos se les conceda un acceso equitativo para evitar inanición (no puede entrar en CPU).
- **Respuesta diferencial:** Ante procesos diferentes y con distinta prioridad, el SO puede necesitar discriminar recursos y tiene que tomar decisiones de forma dinámica.
- **Eficiencia:** El SO debe maximizar la productividad (procesos terminados por unidad de tiempo), minimizar el tiempo de respuesta (unidad de tiempo que tarda un proceso en ser atendido). Estos criterios entran en conflicto, por tanto, es importante medir la actividad del sistema.

Los principales elementos del SO para llevar a cabo los tres factores de equitatividad, respuesta diferencial y eficiencia son:

- **Planificador:** Planificación de procesos y asignación de recursos en un entorno de multiprogramación (intercambio de procesos en CPU con un núcleo).
- **Manejador de llamadas nativas al sistema:** La gestión adecuada de estas llamadas que son funciones propias del SO llamadas a partir de las APIs a nivel de usuario.

- **Manejador de interrupciones:** Estas son provocadas por dispositivos de E/S (dato que espera proceso) se deben gestionar de la mejor manera posible, invirtiendo el menor tiempo posible.

El SO mantiene un número de colas en cuanto a planificación, que son listas de procesos esperando recursos:

- **Cola a corto plazo:** Compuesta por procesos que se encuentran en memoria principal y están listos para ejecutar si el procesador está disponible.
- **Cola a largo plazo:** Lista de nuevos trabajos esperando a utilizar el procesador, pero que no están cargados por completos en memoria principal hasta que no pasan a la cola a corto plazo. El SO debe estar seguro de no sobrecargar la memoria principal ya que debe añadir trabajos al sistema para transferir de largo plazo a corto.
- **Colas de E/S:** Son por cada dispositivo de E/S ya que más de un proceso puede solicitar el uso del mismo dispositivo. Todos los procesos que esperan utilizar los dispositivos alineados en la cola. El SO debe determinar a qué proceso le asigna cada dispositivo disponible.

1.2 EL SISTEMA OPERATIVO COMO INTERFAZ SIMPLE INTERMEDIARIA

La utilización directa del hardware es una tarea difícil para un programador, sobre todo para las operaciones de E/S. Para operar con el controlador de un CD-ROM hay que manejar del orden de 16 órdenes, y donde cada una puede tener más de una docena de parámetros, que hay que empaquetar en un registro del dispositivo. Hay que verificar si el motor está en funcionamiento, si no lo está da la orden de arranque y después da la orden necesaria.

El programador desea una abstracción sencilla y fácil de entender. Una de las funciones del SO es presentar al usuario una máquina que es más fácil de programar que el hardware.

Una máquina virtual es aquella que presenta una mayor facilidad de uso de esta incluyendo toda su funcionalidad. Un usuario ve una abstracción del hardware que logra entender a partir de esta interfaz intermedia. Por ejemplo, llamadas a funciones de lectura o escritura en un disco duro a partir del conjunto de funciones o API de un lenguaje de programación de alto nivel.

El SO debe proporcionar los siguientes servicios:

- **Creación de nuevos programas:** Existen otros programas como los depuradores que no son parte del SO que son necesarios para la creación de nuevos programas.
- **Ejecución de programas:** Para esto, se tiene que realizar una serie de funciones previas como cargar el código lo tiene que gestionar el SO.
- **Operaciones de E/S:** Un proceso puede requerir una operación de E/S. En el ejemplo anterior del CD-ROM, el SO es el encargado de hacer las funciones que permiten la lectura, escritura... Esto es transparente al usuario.

- **Manipulación y control del sistema de archivos:** El SO debe conocer la propia estructura y formato de almacenamiento del periférico y proporcionar los mecanismos adecuados para su control y protección.
- **Detección de errores:** Hay una gran cantidad de errores como los de segmentación, violación de permisos... que el SO debe ser capaz de detectarlos y solucionarlos o hacer que tengan el menor impacto posible.
- **Control de acceso al sistema:** En sistemas públicos, el SO debe controlar quién tiene acceso y a qué recursos(permisos). Por esto, debe tener mecanismos de protección de los recursos e implementar una política de seguridad para que no acceda quién no esté autorizado.
- **Elaboración de informes estadísticos:** Este tipo de informes para conocer el grado de utilización de recursos, configuraciones... puede ser usado incluso por el planificador del sistema.

2. DISEÑO MODULAR EN NIVELES

Con un método de diseño modular las características y la funcionalidad globales del sistema se separan de esta manera:

- Mayor libertad de cambio.
- Menor esfuerzo de mantenimiento.
- Mayor facilidad de correlación de errores y depuración.
- Menor esfuerzo de ampliación del sistema.

Es importante ocultar los detalles a "ojos" de los niveles superiores, ya que ~~deja~~^{robar} libertad a los programadores siempre que la interfaz externa de la rutina permanezca invariable y la propia rutina realice la tarea anunciada. Es decir, cada nivel oculta a los superiores la existencia de determinadas estructuras.

3. TRANSFORMACIÓN DE UN PROGRAMA EN INSTRUCCIONES MÁQUINA

Las máquinas interpretan las señales on y off, lo que equivale a 1 y 0 respectivamente. Las instrucciones son colecciones de bits que pueden ser vistos como números, pero al esto ser complicado y tedioso surgen los lenguajes de alto nivel. Para que los computadores puedan procesar los diferentes tipos de lenguajes se desarrollan otros programas que los traducen, los compiladores.

De un lenguaje de alto nivel, pasa a otro lenguaje, el ensamblador que está a más bajo nivel. Este a su vez es otro programa que traduce un fichero de ensamblador a objeto que está en código máquina (binario).

El código máquina está formado por un conjunto de instrucciones que determinan una serie de acciones que debe realizar la máquina. Ejemplo: "add A, B" se traduce como "1000110010100000". Hay un ensamblador para cada tipo de procesador, pero todo está muy estandarizado.

3.1 EJEMPLO DE PASO DE LENGUAJE ENSAMBLADOR A LENGUAJE MÁQUINA

En ensamblador MOV AL 61h asigna el valor hexadecimal 61 al registro AL. El ensamblador lo pasa a binario: 101100001100001, que el código máquina lo divide en dos: 10110000 y 01100001 donde el primero contiene la instrucción MOV y el código de registro donde se va a mover el dato (operación). En el segundo, se especifica el numero 61h escrito en binario que asignará al registro AL (dato).

Aquí el funcionamiento del SO está codificado a nivel de instrucciones en código máquina que se carga al arrancar la computadora para proporcionar servicios y gestionar recursos para los procesos. Este código se ejecuta en modo núcleo siendo el diseñador el encargado de hacer modificaciones a más bajo nivel. Esto da lugar al modo usuario y modo núcleo.

4. ELEMENTOS BÁSICOS Y ORGANIZACIÓN DE UN COMPUTADOR

Al más alto nivel (arquitectura de Von Neumann), un computador digital consta de:

- **Unidad de procesamiento (CPU):** Controla el funcionamiento del computador y realiza sus funciones de procesamiento de datos. La CPU contiene:
 - Unidad aritmeticológica (ALU)
 - Registro de instrucciones (IR): Instrucción que tengo que ejecutar.
 - Contador de programa (PC): Dir. de memoria de próxima instrucción.
 - Registros auxiliares (RDIM, RDAM, RDI E/S, RDA E/S)
- **Memoria principal:** También llamada memoria real o primaria, es volátil y consta de un conjunto de posiciones definidas mediante direcciones numeradas secuencialmente. Cada posición contiene un patrón de bits que se puede interpretar como una instrucción o como datos.
- **Módulos o controladores de entrada y salida E/S:** Es una parte del hardware que transfiere los datos entre el computador y su entorno externo y viceversa. El entorno externo está formado por dispositivos de memoria secundaria, terminales... Además, contiene buffers que mantiene temporalmente los datos hasta que se puedan enviar.

Todos los componentes se interconectan para que se pueda lograr la función principal del computador, ejecutar programas. Para esto se usan los buses del sistema, que proporcionan comunicación entre los procesadores, memoria P. y módulos E/S.

La CPU y las controladoras de dispositivos pueden funcionar de forma concurrente MENIE compitiendo por la memoria principal. Para asegurar el acceso a esta de forma ordenada, se proporciona una controladora de memoria cuya función es sincronizar el acceso a la misma.

Las placas base modernas suelen incluir dos chips o circuitos integrados para la comunicación entre componentes de la placa denominados puente norte y puente sur. La controladora de memoria puede ser un chip independiente o estar integrado en la CPU.

4.1 EL PROCESADOR Y SUS REGISTROS

Una función del procesador es el intercambio de datos con la memoria y los dispositivos de E/S a través de su módulo. Para esto se usa un conjunto de registros de la CPU:

- **Registro de dirección de memoria (RDIM o MAR):** Almacena temporalmente una dirección de memoria principal en la que escribir o de la que leer.
- **Registro de datos de memoria (RDAM o MDR):** Almacena temporalmente datos que se reciben de la memoria principal o que se pueden escribir en ella.
- **Registro de dirección de E/S (RDIE/S):** Almacena temporalmente una dirección de memoria de los buffers del módulo de E/S en la que escribir o de la que leer.
- **Registro de datos de E/S (RDAE/S):** Almacena temporalmente datos que se reciben de los buffers del módulo de E/S o que se pueden escribir en ellos.

Hay otros registros necesarios para la ejecución de instrucciones:

- **Registro de instrucción (IR):** Contiene la última instrucción leída y que hay que ejecutar, es decir, el contenido de la última dirección de memoria cargada.
- **Contador de programa (PC):** Contiene una dirección de memoria a cargar, la de la próxima instrucción que se cargará desde la memoria principal. La CPU actualiza el PC después de cada captación de instrucción en el registro IR.
- **Palabra de estado del programa (PSW):** Contiene un bit para habilitar/inhabilitar las interrupciones, un bit de modo usuario/supervisor (cambio núcleo/usuario) y bits de códigos de condición (indicadores) cuyo valor asigna el hardware. Por ejemplo, una operación aritmética puede producir un resultado positivo, negativo, cero o desbordamiento aritmético.
- **Otro tipo de registros:** Para las tareas de almacenamiento temporal, como por el direccionamiento, control y condiciones de las operaciones que se realizan, por ejemplo:
 - **Acumulador (AC):** Almacena resultados de cálculos matemáticos de la ALU.
 - **Registros de propósito general (AX, BX, CX, DX):** Almacena bits de datos o instrucciones.
 - **Punteros de pila:** Para el proceso que esté en ejecución habrá un registro dedicado a señalar la cima de la pila y otro para la base.

Estos se pueden clasificar en función de si son visibles o no para el usuario:

- **Visibles para el usuario (modo usuario):** Se permite su acceso por parte del programador mediante lenguaje máquina o en ensamblador. Dependiendo de la arquitectura del procesador, estos registros pueden cambiar.
- **De control y estado no visibles para el usuario (modo núcleo):** Usados por el procesador para controlar su operación y por rutinas privilegiadas del SO para controlar la ejecución de programas. Acceso por instrucciones de máquina.

4.2 INTERACCIÓN DE LA CPU CON LA MEMORIA PRINCIPAL

Los programas de la computadora deben hallarse en la memoria principal para ser ejecutados. La memoria principal es el único área de almacenamiento de gran tamaño a la que el procesador puede acceder directamente. La interacción con esta memoria se consigue a través de:

- Una secuencia de carga desde memoria a CPU: La instrucción **load** mueve una palabra desde la memoria principal a un registro interno de la CPU (IR).
- Una secuencia de almacenamiento de instrucciones desde CPU a memoria: La instrucción **store** mueve el contenido de un registro de la CPU a la memoria principal.

Los programas y los datos deberían estar en la memoria principal de forma permanente pero no es posible por las siguientes razones:

- La memoria principal es demasiado pequeña para los programas y los datos.
- La memoria principal es volátil y pierde su contenido al quitar la alimentación.

La mayor parte de los sistemas operativos proporcionan almacenamiento secundario para poder almacenar grandes cantidades de datos de forma permanente. El más común es el disco duro SSD. Otros son los CD-ROM, pendrives, tarjetas SD...

4.3 ESTRUCTURA DE E/S: DRIVERS Y CONTROLADORES

Gran parte del código del SO se dedica a gestionar la entrada y salida de datos debido a la variada naturaleza de los dispositivos, que son muy lentos comparados con la velocidad de la CPU. Cada controladora de dispositivo se encarga de un tipo específico de dispositivo.

Una controladora de dispositivo mantiene algunos búferes locales y un conjunto de registros de propósito especial. Es la responsable de transferir los datos entre los dispositivos periféricos que controla, su búfer local y la CPU.

Los SO tienen al menos un controlador/a de dispositivo software (driver) para cada controlador/a de dispositivo hardware (controller). Lo importante es no confundir la controladora de dispositivo hardware con el controlador/a del software.

El driver y la controladora de dispositivos funcionan así:

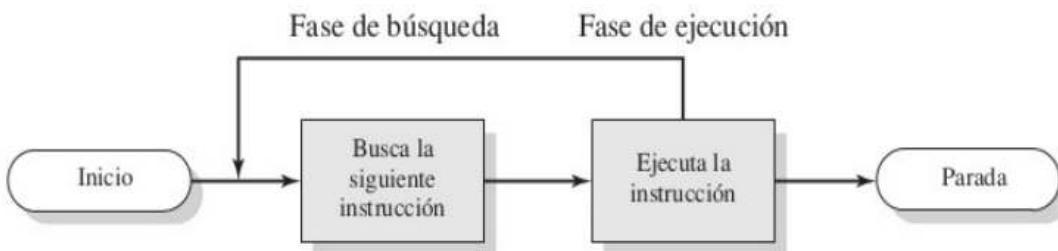
1. Al iniciarse una operación de E/S, la CPU carga el driver.
2. El driver carga los registros apropiados.
3. La controladora examina el contenido de estos registros para determinar qué acción se le ha comunicado.
4. Esta avisa a la controladora de su finalización.
5. La controladora informa a la CPU mediante un envío de señal de interrupción de que ha terminado la operación.
6. La CPU carga el driver.
7. El driver adapta los datos de la controladora, lo devuelve a la CPU y ésta a la RAM.
8. Para otras operaciones, el driver devuelve solo información de estado (OK).

5. BÚSQUEDA Y EJECUCIÓN DE INSTRUCCIONES

Un programa consta de un conjunto de instrucciones almacenado³⁸ en memoria principal. El procesamiento de cada una consta de dos pasos: el procesador lee instrucciones de la memoria, de una en una y las ejecuta.

Desde el punto de vista de los registros, el procesador siempre incrementa el PC después de cada instrucción cargada en el IR, de manera que se leerá la siguiente instrucción en orden secuencial (la instrucción situada en la siguiente dirección de memoria) a no ser que haya instrucciones de salto. Ejemplo: si el procesador lee una instrucción en la posición 300, sino hay salto después leerá la 301,302... Si las hubiera como por ejemplo si en la posición 149 se especifica que la siguiente instrucción es la 182, el procesador almacena en el contador el valor de la 182.

Se denomina **ciclo de instrucción** al **procesamiento requerido por una única instrucción y formado por dos pasos llamados fase de búsqueda y fase de ejecución**.



La ejecución en un sistema se detiene sólo si se apaga la máquina o se produce un error irrecuperable.

Si se está ejecutando un proceso se termina si se ejecuta una instrucción que haga salir al proceso de la CPU.

5.1 EJEMPLO DE EJECUCIÓN DE INSTRUCCIONES

Características del ejemplo:

- El procesador tiene un único registro de datos (AC), más el registro IR y PC.
- Las instrucciones y los datos tienen una longitud de 16 bits.
- La instrucción tiene 4 bits para el código de operación permitiendo $2^4 = 16$ códigos de operación diferentes.
- Con los 12 restantes se pueden direccionar hasta $2^{12} = 4.096$ (4K) palabras de memoria.

El programa hace $941 \leftarrow 940 + 941$ (suma de los dos y lo almacena en la 941). Para esto se necesita 3 instrucciones (3 fases de búsqueda y 3 de ejecución):

1. El PC contiene el valor 300 (1^a instrucción) que es el valor 1940 en hexadecimal, se carga dentro del IR y se incrementa el PC. El 1 indica que en el AC se va a cargar un valor leído de la memoria.
2. Se carga en el AC el contenido de la 940 (0003) en hexadecimal.
3. La lee la 5941 de la 301 y se incrementa el PC.
4. Se suman los contenidos en el AC.
5. Se lee la 2941 de la 302 y se incrementa el PC.
6. Se almacena el contenido del AC en la posición 941.

6. INTERRUPCIONES Y MANEJADOR DE INTERRUPCIONES

El procesador puede verse interrumpido por un suceso o evento. Esta se indica mediante una interrupción hardware o software.

Los tipos de interrupciones hardware son:

- **Interrupciones de E/S:** Generada por un controlador/a de dispositivos de E/S para señalar la conclusión normal de una operación o para indicar condiciones de error.
- **Interrupciones por fallo de hardware:** Se generan por cortes en el suministro de energía o zonas corruptas de memoria.

Los tipos de interrupciones software son:

- **Interrupciones por temporizador:** Se generan por un temporizador del procesador y permiten al SO realizar ciertas funciones de forma regular.
- **Interrupciones de proceso (excepciones):** Generadas por alguna condición que se produce como resultado de la ejecución de una instrucción. Las excepciones no son interrupciones como tales, es una instrucción provocada por una situación de error detectada por la CPU mientras ejecutaba una instrucción.
Hay instrucciones que si se producen en programas a nivel de usuario pueden controlarse de forma que el programa que la causa puede continuar con su ejecución.

Cuando se interrumpe a la CPU ésta deja lo que está haciendo y transfiere la ejecución a una posición de memoria principal fija y establecida que contiene la dirección de inicio de la rutina de servicio a la interrupción (ISR) que se ejecuta, se trata la interrupción y cuando ha terminado se le devuelve el control a la CPU, la cual reanuda la operación o proceso que estuviera ejecutando antes de la interrupción o se finaliza si es una excepción.

La interrupción debe transferir el control a la ISR apropiada de tal manera:

- **Método 1:** El más simple es la que la transferencia consiste en invocar a una rutina ISR genérica para examinar la información de la interrupción; esa rutina genérica llama a la específica de tratamiento de la interrupción según cual sea el origen que la produce.
- **Método 2:** Consiste en disponer de una tabla de punteros a las diferentes rutinas de interrupción (vector de interrupciones) para proporcionar velocidad. Es una tabla o matriz de la memoria principal, donde cada elemento contiene la dirección de memoria de las diferentes rutinas ISR que se pueden ejecutar.

El vector de interrupciones se indexa mediante un número de interrupción único, que se proporciona en la propia solicitud de interrupción, por medio de una controladora hardware, y sirve para obtener la dirección de la ISR específica para el tratamiento de la interrupción concreta que se ha producido.

Es muy importante saber que cuando se invoca a una ISR se almacena la dirección de la instrucción interrumpida correspondiente al proceso actual. Después de

atender a la interrupción y si se decide así por la política de planificación en uso, la dirección de retorno guardada se carga en el contador de programa y los registros salvados se vuelven a cargar para que el programa se reanude, esto se llama “salvado y restauración de contexto”.

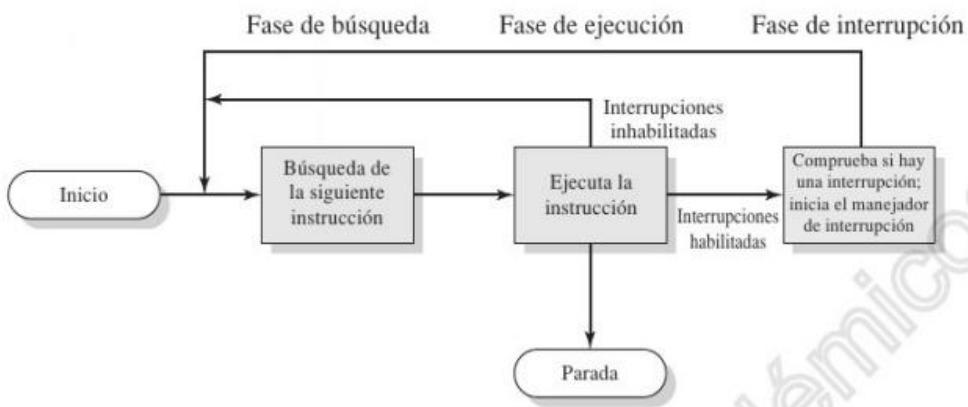
Cuando estamos hablando de interrupción el procesador está realizando una tarea concreta. Invocar una operación de E/S no es una interrupción, aunque tenga que salvarse para ejecutar la llamada nativa.



6.1 ADICIÓN DE LA FASE DE INTERRUPCIÓN

El programa de usuario no tiene que contener ningún código especial para tratar las interrupciones, el procesador y el SO son responsables de suspender el programa de usuario y, posteriormente reanudarlo en el mismo punto.

Para esto, es necesario añadir una fase de interrupción al ciclo de instrucción:



En la fase de interrupción el procesador comprueba si se ha producido cualquier interrupción que es indicado por la presencia de una señal y enviada por la controladora de un dispositivo de E/S mediante un bus del sistema.

En el caso de que no haya interrupciones pendientes después de la comprobación, el procesador continúa con la fase de búsqueda y lee la siguiente instrucción.

6.2 INTERRUPCIONES MÚLTIPLES

Se consideran 2 alternativas a la hora de tratar con múltiples interrupciones:

1. Inhabilitar las interrupciones mientras se está procesando una interrupción. Esto quiere decir que el procesador ignorará cualquier nueva señal de petición de interrupción. Después de que se completa la rutina, se rehabilitan las interrupciones antes de reanudar el programa de usuario y el procesador comprueba si se han producido interrupciones adicionales para atenderlas. La desventaja de la estrategia anterior es que no tiene en cuenta la prioridad relativa o el grado de urgencia de las interrupciones.

2. Definir prioridades para las interrupciones y permitir que una interrupción de más prioridad causa que se interrumpa la ejecución de un manejador de una interrupción de menor prioridad.

Como desventaja, el sistema puede dejar en inanición a otras interrupciones. Para controlar eso está el planificador de dispositivos de E/S, para que no se sirva de manera infinita a interrupciones de prioridad alta.

7. SISTEMAS DE UN SOLO PROCESADOR

La presencia de microprocesadores de propósito especial resulta bastante común y nos convierte a un sistema de un solo procesador en un sistema multiprocesador. Si sólo hay una CPU de propósito general, entonces el sistema es de un solo procesador.

8. SISTEMAS MULTIPROCESADOR

Los procesadores se comunican entre sí compartiendo el bus de la computadora, la memoria y los dispositivos periféricos.

La ingeniería multinúcleo fue impulsada por la fuga de corriente y la cantidad de calor generada por el chip. Esta mejora tecnológica tiene un problema, y es que el SO debe tener mecanismos que permitan trabajar con varios núcleos y/o varios procesadores.

Los sistemas multiprocesador tienen ventajas fundamentales:

- Mayor rendimiento: Se realizan más trabajos independientes en menos tiempo.
- Economía de escala: Los sistemas multiprocesador pueden resultar más baratos que su equivalente con múltiples sistemas de un solo procesador ya que pueden compartir periféricos, fuentes de alimentación...

8.1 SISTEMAS MULTIPROCESADOR SIMÉTRICOS Y ASIMÉTRICOS

Pueden ser simétrico o asimétrico:

1. Asimétrico: Cada procesador se asigna a una tarea específica. Un procesador maestro controla el sistema y el resto de los procesadores esperan que el maestro les dé instrucciones. Este esquema define una relación maestro-esclavo. La ventaja es su simpleza, aunque, presenta desventajas serias como:

- Confiabilidad: no es mayor que la de un sistema con procesador único, ya que si falla el maestro falla todo el sistema.
- Uso deficiente de los recursos: si un procesador esclavo se libera mientras el maestro está ocupado, tiene que esperar a que el maestro acabe.
- Interrupciones: todos los procesadores esclavos deben interrumpir al maestro cada vez que requieran atención.
- Baja disponibilidad: ante una gran carga de trabajo del maestro, ya que haría que no se pudieran atender en un tiempo adecuado nuevos procesos o tareas.

2. **Simétrico:** El multiprocesamiento simétrico (SMP) en el que todos los procesadores son iguales y tiene las siguientes características:

- Comparten las mismas utilidades de memoria principal y de E/S.
- Todos los procesadores pueden realizar las mismas funciones.
- La existencia de múltiples procesadores es transparente al usuario.

Tiene más ventajas sobre las arquitecturas monoprocesador, además de mayor rendimiento y economía de escala, presenta las siguientes ventajas:

- **Confiable:** si un procesador deja de funcionar no se cae el sistema entero.
- **Disponibilidad:** es la capacidad de un servicio o de un sistema a ser accesible y utilizable por los usuarios o programas autorizados cuando estos lo requieran.

Estas características son beneficios potenciales, no garantizados. El SO debe proporcionar herramientas y funciones para explotar el paralelismo en un sistema SMP.

9. MULTIPROGRAMACIÓN

La multiprogramación incrementa el uso de la CPU de manera que ésta no quede ociosa de trabajos. La idea es:

El SO mantiene en memoria principal, de forma simultánea, varios trabajos, y empieza a ejecutar uno de ellos. El trabajo puede tener que esperar a que se complete alguna otra tarea como una operación de E/S. Si no fuera multiprogramado, la CPU se quedaría inactiva hasta que una operación de E/S concluyese o el evento esperado se produjese.

En un sistema multiprogramado, el SO simplemente cambia el trabajo y ejecuta otro mientras que se realiza esa operación E/S o se espera a ese evento concreto. A qué trabajo se cambia es decisión del planificador que el sistema utilice.

Un ejemplo de multiprogramación en un sistema con un solo procesador, pero simulando que en memoria principal hay 1, 2 o 3 procesos listos para ejecución y que realizan operaciones de E/S.

10. MULTIPROCESAMIENTO

Con la multiprogramación, sólo un proceso se puede ejecutar a la vez, mientras el resto esperan por el procesador. Con multiproceso, más de un proceso puede ejecutarse simultáneamente, cada uno de ellos en un procesador distinto, pero en un sistema con multiprocesamiento también hay multiprogramación.

10.1 HYPERTHREADING

Permite ejecutar dos hilos dentro de un único núcleo del procesador. Los procesos pueden estar formados por un solo hilo de ejecución o por varios hilos.

Esta tecnología consiste en simular en cada núcleo físico del procesador dos procesadores lógicos, dividiendo los hilos entre ambos y mejorando la velocidad del procesamiento. El resultado es una mejoría en el rendimiento del procesador gracias a la duplicación en la CPU de los registros de propósito general, registros de control como el PC o el IR. Los SO deben estar preparados para esta tecnología, pero los modernos ya lo implementan en sus núcleos al soportar modelos multihilo o SMT y multiprocesamiento simétrico o SMP.

HyperThreading conlleva una mejora en la velocidad de las aplicaciones, que es de un 30% más comparado con un procesador que no lo implemente.

Existe paralelismo real cuando dentro de un núcleo, los hilos de ejecución que haya no tengan que utilizar a la vez aquellas partes hardware replicadas, ya que en un núcleo con HyperThreading hay registros duplicados (la ALU no se replica y debe ser compartida).

Cada núcleo puede realizar multiprogramación, y cuando un núcleo usa HyperThreading también usa “multiprogramación” ya que ciertas unidades de cálculo deben compartirse entre los hilos.

Los hilos pueden pertenecer a procesos monohilo diferentes o a dos hilos de un mismo proceso multihilo.

HyperThreading es beneficioso cuando la carga de trabajo requiere un procesamiento de tareas que se pueden realizar en paralelo. También es útil cuando se desea que la CPU ejecute tareas más ligeras mientras que las intensivas se envían a otro núcleo en procesadores de múltiples núcleos.

11. MODO DUAL (USUARIO Y KERNEL)

El código del propio SO se encuentra codificado en lenguaje máquina al cargarse en memoria principal al arranque del sistema.

Si el usuario a través de un programa de aplicación puede hacer cambios por ejemplo en las rutinas de ISR, se podría hacer un uso indebido del sistema pudiendo corromper el funcionamiento y gestión de los recursos, programas y usuarios del sistema.

Para evitarlo, hay una solución basada en un modo dual de operación:

- **El modo núcleo, kernel, supervisor o privilegiado:** Está asociado al procesamiento de instrucciones y rutinas del SO. Son accesibles sólo a través de llamadas nativas al sistema que son rutinas o funciones a nivel núcleo (situadas en la parte de la memoria principal reservada al SO) y no es accesible por parte del usuario.
El núcleo posee las funciones y servicios del sistema que deben protegerse y residen en lenguaje máquina en memoria principal.
- **El modo usuario:** Está asociado al procesamiento de instrucciones y rutinas de los programas de usuario. Un programa de usuario no puede acceder directamente a posiciones de memoria del núcleo del SO, debe hacerlo por llamadas nativas al sistema.

Se accede al código del núcleo:

1. Cuando se ejecuta una llamada nativa al sistema como fork() o wait(). Estas llamadas dan lugar a que se produzca una interrupción especial de acceso al núcleo.
2. Cuando un dispositivo de E/S provoca una interrupción ya que hay que ejecutar las rutinas del núcleo necesarias para tratar esa interrupción.
3. Cuando hay alguna interrupción por fallo de tipo hardware que no provoque el apagado repentino de la máquina.
4. Cuando el planificador debe cambiar de proceso, p.ej. al cumplir un tiempo de reloj asignado o por realizar funciones de gestión de los recursos del sistema.
5. Cuando una aplicación provoca una excepción, que da lugar a una interrupción software a tratar mediante las rutinas del núcleo necesarias que terminaran con el proceso que provocó la excepción.

Las funciones básicas que se encuentran dentro del núcleo son:

- **Para la gestión de procesos:**
 - Creación y terminación de procesos (fork).
 - Planificación y activación de procesos.
 - Intercambio de procesos.
 - Sincronización de procesos y soporte para comunicación entre procesos.
 - Gestión de los bloques de control de procesos (BCP).
- **Para la gestión de memoria:**
 - Reserva del espacio de direcciones para los procesos a nivel de usuario.
 - Memoria virtual (planificador a medio plazo).
 - Gestión de paginación y segmentación de memoria.
- **Para la gestión de E/S:**
 - Gestión de buffers internos e intermedios.
 - Reserva de canales de E/S y de dispositivos que requieren de su uso por parte de procesos.
 - Manipulación de dispositivos, conectar, desconectar.
 - Comunicaciones, envío y recepción de mensajes, dispositivos remotos.
- **Para la gestión de los sistemas de ficheros:**
 - ext2, ext3, fat32, ntfs...
- **Funciones de soporte:**
 - Gestión de interrupciones, excepciones.
 - Auditoria y monitorización.

Para cambiar de modo existe un bit en la palabra de estado del programa que indica el modo de ejecución, este se cambia como respuesta a determinados eventos como una llamada nativa al sistema o una interrupción.

El hardware detecta los errores de violación de los modos, de tal modo, que, si desde un programa de usuario se hace un intento de ejecutar una instrucción privilegiada del núcleo, el hardware envía una señal de excepción al SO para que la trate y este proporciona un mensaje de error que puede volcarse a memoria (Core dump).

El SO configura periódicamente una interrupción de reloj vía hardware para evitar perder el control y cambiar a modo núcleo frecuentemente.

12. LLAMADAS NATIVAS AL SISTEMA Y PASO DE PARÁMETROS

Se usa para solicitarle al SO la apertura o cierre de un fichero... normalmente se ejecutan miles de llamadas nativas al sistema por segundo. Estas proporcionan un interfaz entre un programa o proceso y el núcleo del SO para poder invocar los servicios que éste ofrece.

Las llamadas al sistema se pueden usar de varias formas:

1. Utilizando una API que es la interfaz de programación de aplicaciones. Esta oculta al programador la mayor parte de los detalles internos de las llamadas nativas al sistema disponibles. Las más usuales son la Win32 para Windows y la de C, glibc para Linux y Mac.

Las funciones que forman una API invocan a estas llamadas a través de una función nombrada TRAP, por lo que actúan como wrappers o funciones nativas del núcleo.

Si se usa printf() en un programa en C, esta no es la llamada al sistema en si, sino que por debajo se encuentra la llamada para realizar la impresión. Lo mismo pasa con write()...

2. En Linux existe una función wrapper denominada syscall() que permite invocar llamadas nativas cuando no nos sirva una función wrapper de una API o no haya función wrapper para ella. Su uso se puede consultar en la terminal con "man 2 syscall". Si se escribe "man man" en la sección 2 está casi todo el conjunto de las llamadas nativas.

Existen más funciones a parte de las 300 ofrecidas por glibc para gestionar los recursos del sistema, pero sólo se ofrecen esas por el SO.

En glibc aunque muchos wrappers tengan nombres similares a algunas llamadas nativas no lo son, son wrappers.

3. En ensamblador la llamada depende directamente del "hardware" sobre el que se está ejecutando el SO (los registros del procesador) y de cómo se cambia de modo (INT 0X80, SYSENTER, SYSCALL).

Cada llamada nativa al sistema tiene asociado un número identificador que hace referencia a la tabla de llamadas al sistema. Esta tabla es un array de punteros a función, se sitúa en la parte de la memoria reservada al núcleo del SO y contiene punteros a llamadas nativas del SO. El índice indica la identificación de la llamada nativa y contiene la dirección de comienzo de esta en memoria.

Cuando se invoca a una llamada nativa se devuelve el estado de la ejecución de dicha llamada nativa + el posible valor de retorno. Esto se hace mediante TRAP.

Al realizarla puede que haya que pasar parámetros al núcleo. Hay dos métodos:

1. Pasar los parámetros a una serie de registros del procesador que son accesibles en modo usuario. Puede haber más parámetros que registros disponibles, entonces los parámetros se almacenan en un bloque en memoria a nivel usuario, y la dirección del bloque se pasa como parámetro a un registro del procesador que después será copiado al núcleo.
2. Colocar los parámetros en la pila de memoria de dicho proceso y el SO se encarga de copiarlos al núcleo.

12.1 PASOS GENERALES EN LA INVOCACIÓN DE UNA LLAMADA NATIVA AL SISTEMA MEDIANTE FUNCIONES DE TIPO TRAP

El proceso general mediante el uso de APIs es:

1. Los parámetros asociados se cargarían en la pila del proceso a nivel usuario, o en registros accesibles. Además, se copia en un registro del procesador el identificador de la llamada nativa. Todavía se está en modo usuario.
2. La propia API invocada ejecuta otra función de tipo TRAP que son las funciones de la propia API las que la invocan internamente. (Modo usuario). La función de tipo TRAP se encargará de ejecutar una interrupción especial que conllevará a un cambio de modo, de usuario a núcleo que se realiza por hardware.
Ya en modo núcleo se copia el identificador de llamada nativa a invocar y los parámetros almacenados por la llamada a nivel de API.
3. Siempre hay que guardar el estado de ejecución actual y el valor del PC. Para ello se siguen invocando otras rutinas justo antes de empezar a ejecutar la llamada nativa al sistema a partir de su número identificación.
4. Se examina el identificador y los parámetros correspondientes a la llamada nativa a invocar y se busca en una tabla si existe el número y los parámetros son correctos.
5. Se procede a ejecutarla. La tabla de llamadas nativas al sistema contiene la dirección del lugar del núcleo donde se encuentra la llamada nativa.
6. Se invocan otras subrutinas llamadas RETURN FROM TRAP:
 - o Devolver el parámetro de retorno que dé la función nativa.
 - o Restaurar el contexto (se copia en la pila lo que hay en el eax).
 - o Pasar de modo núcleo a usuario.
7. Se regresa el control a la API y a nivel de usuario se descarga la pila.

El esquema proporcionado es genérico, una llamada nativa al sistema se podría ver interrumpida por algún evento o señal. Si se programase en ensamblador, el proceso de invocación debe hacerse cuidadosamente. Esto lo suelen hacer diseñadores y programadores del núcleo del SO siendo poco frecuente en el programador habitual. En Linux los ficheros en lenguaje ensamblador tienen extensión .S o .asm.

12.2 EJEMPLO DE LLAMADAS NATIVAS AL SISTEMA EN LINUX CLÁSICO

Vamos a ver cómo se implementan las llamadas en una arquitectura x86 (32 bits). A nivel de lenguaje ensamblador, para una versión 2.6 del núcleo de Linux y arquitectura x86, se utiliza la invocación de una interrupción especial INT 0x80 como manera de comenzar el proceso de acceso a una llamada nativa al sistema y cambiar a modo núcleo. Los más modernos utilizan SYSCALL o SYSENTER.

En unistd.h se encuentran identificadas todas las llamadas nativas al sistema con su correspondiente número.

Int 0x80 es la interrupción software que iniciará el proceso de acceso a la llamada nativa que se vaya a invocar. Este está definido por la constante SYSCALL_VECTOR. En el código de la función __init trap_init() el método de entrada al núcleo es mediante la función:

```
Set_system_gate(SYSCALL_VECTOR, &system_call)
```

En la llamada a esta función se dan los siguientes pasos:

1. system_call() guarda en la pila el código identificador de la llamada nativa al sistema recogido del registro %eax y se copian los parámetros de entrada.
2. Se guardan varios registros del procesador con SAVE_ALL.
3. Se comprueba que el número de llamada nativa pedido es válido. Si no es válido, se ejecuta el código “syscall_badsys” que devuelve el código de error ENOSYS para ponerlo en errno y %eax se pone a -1 para saltar a “resume_userspace()”.
Si es válido se ejecuta el código bajo la etiqueta “syscall_call” que se invoca a la llamada nativa mediante sys_call_table() y se guarda el valor de retorno en %eax.
Al finalizar se ejecuta el “syscall_exit”. Si no hay nada más prioritario se ejecuta el código situado bajo “syscall_exit_work” que terminará saltando a resume_userspace().
4. Para poder restaurar y seguir a partir de la llamada al sistema a nivel de API se ejecuta resume_userspace() que ejecuta “restore_all” que se restauran los registros almacenados previamente con SAVE_ALL y se cambia a modo usuario mediante instrucciones de tipo IRET.
5. Si la llamada fue exitosa glibc obtiene el resultado de la pila y seguirá la ejecución del proceso de usuario por donde iba.
6. Si no tuvo éxito o existe algún error el valor devuelto a %eax es -1 y habría que consultar la variable errno.

12.3 LOCALIZAR CUÁLES SON LOS NÚMEROS (IDs) DE LLAMADAS AL SISTEMA EN UN SO LINUX ACTUAL

El mecanismo de implementar las llamadas varía según la arquitectura de la máquina y su versión del kernel. Se puede ver el tipo de información sobre nuestro sistema con los comandos:

- Uname -a para la versión del kernel.
- Lsb_release -cdir para la versión del SO.
- Lscpu para información de la CPU.

Para encontrar el fichero unistd.h se usa el comando locate junto a unistd.h y veremos una lista de todos los ficheros con ese nombre en nuestro sistema.

- /usr/include/asm-generic/unistd.h es donde está la lista de las llamadas nativas al sistema.

13. INTÉRPRETE DE COMANDOS

Los intérpretes se conocen como shells. La función es obtener y ejecutar un comando especificado por el usuario. Muchos se utilizan para manipular archivos: creación, borrado...

El intérprete de comandos no “entiende” el comando, sino que lo usa para identificar el archivo que hay que cargar en memoria y ejecutar.

TEMA 2: PROCESOS E HILOS

1. PROCESOS Y BLOQUE DE CONTROL DE PROCESOS (BCP)

Un programa o tarea es una unidad inactiva, no es un proceso. Hay muchas definiciones de proceso:

- Un programa en ejecución.
- Una instancia de un programa ejecutándose en un computador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad cargada en memoria principal y caracterizada por un hilo secuencial de ejecución, un estado actual, y un conjunto de datos y recursos del sistema asociados al proceso (contexto de un proceso).

El contexto de un proceso es un conjunto de datos por el que el SO es capaz de supervisar el proceso y controlarlo. Se hace en una estructura de datos que se llama (Bloque de Control de Proceso (BCP)) que está almacenado en una lista simplemente enlazada, llamada tabla de procesos.

Algunos datos destacables son:

- **Identificación:** Un identificador único asociado al proceso para distinguirlo del resto de procesos.
- **Estado:** Si el proceso está corriendo, está en el estado de Ejecución. Existen estados como Listo, Bloqueado, Suspenido, Terminado o Zombie.
- **Información de planificación:** Nivel de prioridad relativo al resto de eventos por los que espera el proceso cuando está Bloqueado. Se usa para indicar qué trabajo se realiza a continuación.
- **Descripción de los segmentos de memoria asignados al proceso:** Espacio de direcciones asignado al proceso en el espacio de usuario en RAM.
- **Punteros a memoria:** Punteros al código del programa y los datos asociados dicho proceso. Incluye los límites del último registro de activación de la pila.
- **Datos de contexto en relación con los registros de la CPU:** Son datos que están presentes en los registros del procesador cuando el proceso está corriendo (todo lo necesario para poder continuar la ejecución del proceso).
- **Información de estados de E/S y recursos asignados:** Incluye las peticiones de E/S pendientes...
- **Comunicación entre procesos:** Se guarda cualquier dato que tenga que ver con la comunicación entre procesos independientes, información de:
 - Memoria compartida.
 - Señales.
 - Semáforos.
 - Paso de mensajes.
 - Tuberías y colas.
 - Paso de mensajes en red.
 - Sockets.
 - Estándar MPI (implementación del std que proporciona librerías para comunicar procesos entre máquinas en red).
- **Información de auditoría:** Ciclos utilizados por el proceso (lo que queda).

Es posible interrumpir un proceso cuando está corriendo y posteriormente restaurar su estado de ejecución. Por tanto, el BCP es la herramienta clave que permite proporcionar multiprogramación.

Un BCP se crea cuando se acepta un proceso en el sistema por parte del planificador a largo plazo, y se actualiza a medida que la ejecución del proceso avanza.

Se llama imagen del proceso al conjunto del código y BCP de un proceso. Para ejecutar un proceso es necesario que su imagen esté cargada en memoria principal.

2. DISPATCHER

Para un monoprocesador un único proceso puede estar ejecutándose en un instante de tiempo, y dicho proceso estará en el estado “Ejecución”. La conmutación entre unos procesos y otros da lugar a la multiprogramación y el proceso que se encarga de ello se llama activador o dispatcher, el cual reside en el núcleo del sistema.

Este pasa a ejecución cuando se producen:

- Interrupciones:
 - Por temporización (rodaja o quantum de tiempo).
 - La llegada de un proceso prioritario o interrupciones software.
- Solicitud de operaciones bloqueantes: Invocar por ejemplo, wait().
- Solicitud de operaciones de E/S.

El planificador a corto plazo es el que dice al dispatcher qué proceso es el que debe introducir en la CPU. Vamos a tratar por un lado al planificador a corto plazo y por otro al dispatcher, aunque a veces se nombre como que es lo mismo. El planificador a corto plazo se encargará de reordenar la lista de procesos listos para su ejecución y el dispatcher recogerá al primero según ese ordenamiento.

Gracias al BCP, el dispatcher puede introducir un proceso nuevo o hacer que se continúe con otro por la instrucción donde se quedó (salvado y restauración de contexto).

Una interrupción puede suceder en cualquier momento y en cualquier punto de la ejecución. Su aparición es imprevisible.

2.1 EJEMPLO DE EJECUCIÓN DISPATCHER

Suponiendo 3 procesos en memoria principal que representan 3 programas y, además:

- Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C y las 4 del B.
- El activador es un pequeño programa con 6 instrucciones.
- El SO sólo deja que un proceso continúe durante 6 ciclos de instrucción (rodaja de tiempo), después se interrumpe por una alarma de reloj. Previene que se provoque inanición en los demás procesos.
- Se omite:
 - La ejecución de la rutina ISR
 - La ejecución del proceso del planificador a corto plazo.

- Se supone que el planificador decide comenzar a ejecutar el proceso A.

Las trazas desde el punto de vista del procesador son:

A → Dispatcher → B → Dispatcher → C → Dispatcher → A → Dispatcher → C

La ejecución se resume de la siguiente manera:

1. Las primeras 6 instrucciones del proceso A se ejecutan seguidas de una alarma de temporización y de la ejecución del activador, que ejecuta sus 6 instrucciones antes de devolver el control al proceso B.
2. Despues de ejecutar 4 instrucciones de B, el procesador deja de ejecutarlo y se ejecuta el activador.
3. Se pasa a ejecutar el proceso C y despues se vuelve a ejecutar el activador.
4. El procesador vuelve al proceso A y despues vuelta al activador.
5. Se pasa de nuevo al proceso C.

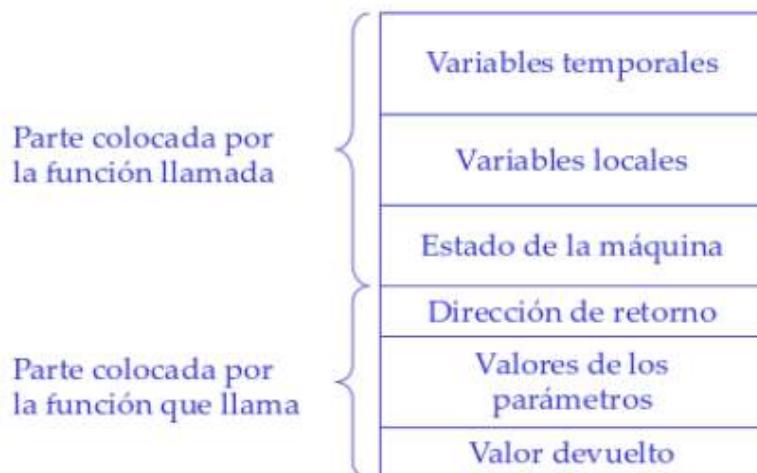
3. CONTROL DE LLAMADAS A FUNCIONES MEDIANTE PILA

Una pila es un conjunto ordenado de elementos llamados registros de activación, que almacenan determinada información sobre las rutinas que se invocan. Se conoce también a una pila como una lista donde el último que entra es el primero que sale.

En la pila se almacenará lo siguiente:

- La dirección de retorno: Dirección de memoria de la siguiente instrucción de código donde retornar después de ejecutarse la función o subrutina.
- Datos pasados como parámetros.
- Valor de retorno: Valor devuelto por la función o subrutina.
- Variables locales.
- Variables temporales.
- Estado de la máquina.

Para cada función o subrutina se apila un registro de activación que es lo siguiente:



Cuaderno Un proceso se compone de varias zonas de memoria principal diferenciadas. Una vez se ha reservado las zonas de memoria, el proceso está listo para ser ejecutado.

- **La pila de llamadas (stack):** Zona de memoria variable para las llamadas a subrutinas, que almacena los registros de activación de llamadas a funciones o subrutinas.
Sirven para que una subrutina pueda acceder a los parámetros y variables que necesita, de forma que se tenga bien localizado su registro de activación.
 - Puntero de pila o **Stack Pointer (SP):** Dirección de la cima de la pila.
 - Puntero de base o **Frame Pointer (FP):** Dirección base del último registro de activación apilado.
- **El área de datos dinámicos o montón (heap):** Zona para la gestión dinámica de la memoria que se solicita durante la ejecución, es una zona de memoria variable (malloc(), calloc(), free()...)
- **El área de datos estáticos:** Se usa para almacenar las variables globales, las constantes y las variables estáticas reservando espacio justo.
- **El área del código:** Almacena el código fuente en instrucciones máquina. Se reserva solamente el espacio justo.

El conjunto de pasos en una llamada a una rutina y su retorno es:

Pasos en la llamada a una rutina:

- Se reserva espacio para el valor devuelto, pero no se asigna valor alguno aún.
- Se reserva espacio y almacena el valor de los parámetros que se pasan.
- Se reserva espacio y almacena el valor de la dirección de retorno.
- Se reserva espacio y almacena el estado de la máquina, SP y FP.
- Se reserva espacio y almacena las variables locales de la función llamada.
- Comienza la ejecución del código de la función llamada.

Pasos una vez ejecutada la rutina llamada, retorno:

- Se almacena el valor devuelto en el espacio que había para ello en la pila.
- Se copia este valor devuelto en la variable local de la rutina.
- Se recoge el valor de la dirección de retorno de la función y se devuelve el control a la función que llama.
- Se restaura el contenido del estado del estado de la máquina. Esto modifica el valor de FP y SP a los valores de la rutina llamante.

4. ARRANQUE DEL SISTEMA

Para que una computadora comience a funcionar, es necesario que se produzcan una serie de etapas:

1. Se ejecuta un programa de inicio que se almacena en una memoria ROM o EEPROM y se conoce como firmware o Bios. Si se usa una ROM, para cambiar el sistema de arranque hay que cambiar el chip en sí. El propósito de la Bios es identificar y diagnosticar los dispositivos del sistema. Es ejecutada por la CPU. Actualmente se usa UEFI que es una Bios más rápida y con un sistema de arranque más seguro que evita que se inicien SO no autorizados.

2. El programa puede continuar con la secuencia de arranque. El Bios hace que se comience a ejecutar un programa llamado gestor o cargador de arranque de segunda etapa. Los cargadores de arranque suelen encontrarse en el primer sector (bloque 0) del dispositivo de arranque. Este sector se llama Master Boot Record (MBR). Un disco tiene una partición marcada como de arranque en su sector 0. Se llama disco de arranque o disco del sistema. La CPU busca en ese sector y lo carga en memoria para ejecutar el código.
3. Cuando el gestor de arranque empieza a ejecutarse, explora el sistema de archivos para localizar el kernel del SO y cargarlo en memoria principal. Comienza a ejecutarse el núcleo del sistema que realiza tareas como:
 - Configuración de la memoria principal y paginación.
 - Configuración de los subsistemas de E/S.
 - Establecimiento del manejo de interrupciones.
 - Montar el sistema de archivos del root en /
4. Una vez ejecutado el kernel, el SO comienza a establecer el espacio de usuario. Ejecuta una serie de servicios y scripts:
 - Montaje de otras particiones o discos.
 - Inicialización del servicio de red.
 - Inicio del entorno gráfico.
 - Inicio de sesiones de otros usuarios y control de inicio de sesión.

Como ejemplos de cargadores de arranque tenemos Lilo y Grub en Linux.

5. CREACIÓN Y TERMINACIÓN DE PROCESOS

Hay 2 eventos generales que provocan la creación de procesos:

1. El arranque del sistema: En Linux usando el comando `service -status-all` para visualizar los servicios que se están ejecutando en el sistema. Posterior al arranque también se pueden cargar e iniciar nuevos servicios.
2. La invocación interna desde un proceso de algún ejecutable o invocación de una llamada al sistema para creación de procesos: Esta invocación también se puede producir:
 - Por el uso del ratón haciendo un doble clic para abrir un programa.
 - Mediante la invocación de algún ejecutable desde la terminal.
 - A partir de un script.

Tarde o temprano un proceso terminará debido a:

Salidas voluntarias:

- Salida normal: La mayoría de los procesos terminan debido a que han concluido su trabajo. En sistemas con entorno gráfico se puede hacer clic para indicar al proceso que termine. Esta acción en realidad lleva a invocar a `exit()` tras ejecutarse un listener de eventos, en este caso el recoger la acción de clic del ratón.
- Salida controlada por error: Porque el proceso descubrirá un error. Este error es predecible y tenido en cuenta por el proceso, ya que el programador contempla la opción de terminar el programa.

Salidas involuntarias:

- **Error fatal:** A menudo debido a un error en el programa como:
 - Ejecutar una instrucción ilegal.
 - Hacer referencia a una parte de memoria no existente o no reservada.
 - División entre 0.
- Un proceso puede indicar al SO que desea manejar ciertos errores por sí mismo. Esto forma parte del tratamiento de errores y captura de excepciones. El ANSI C++ especifica que si una instrucción new() falla (no hay memoria disponible) lanza la excepción bad_alloc que se encierra en un bloque try. La instrucción catch() captura la excepción bad_alloc.
- **Eliminado por otro proceso:** Algun otro proceso ejecute una llamada al sistema que indique que lo elimine (kill()). Puede ocurrir la finalización con procesos lanzados desde un terminal. Si se cierra un terminal, éste se encargaría de eliminar a sus procesos hijos.

6. UN MODELO DE ESTADOS PARA PROCESOS

La responsabilidad principal del SO es controlar la ejecución de los procesos y asignar recursos a los mismos. El primer paso es "construir" un modelo de comportamiento para los procesos. Un esquema general de los 5 estados es:



Los 5 estados son:

- **Nuevo:** Se trata de un nuevo proceso no ha sido cargado en memoria principal, aunque su BCP sí ha sido creado por el planificador a largo plazo. El proceso en sí aún no se ha puesto en ejecución, su imagen no está cargada en RAM. Un SO puede limitar el número de procesos que puede haber por razones de rendimiento o limitaciones de memoria principal.
- **Ejecutando:** El proceso está actualmente en ejecución.
- **Listo:** Un proceso que se prepara para ejecutar cuando tenga oportunidad.
- **Bloqueado:** No puede ejecutar hasta que se cumpla un evento determinado.
- **Saliente:** Un proceso que ha sido liberado del grupo de procesos ejecutables por el SO, debido a que ha sido detenido o abortado por alguna razón. La terminación de un proceso lo mueve a Saliente, el proceso no es elegible de nuevo para su ejecución. Las tablas asociadas al proceso se encuentran temporalmente preservadas por el SO que proporciona tiempo para que programas auxiliares extraigan la información necesaria.

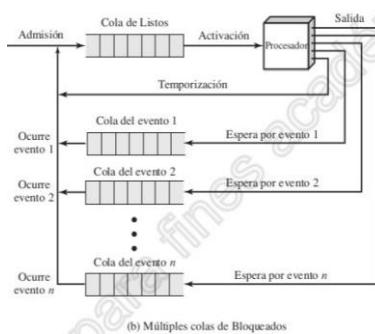
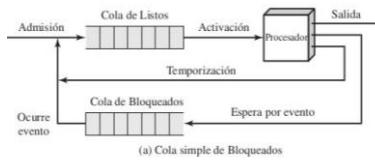
6.1 TRANSICIÓN ENTRE ESTADOS

Las posibles transiciones son:

- **Null → Nuevo:** Se crea un nuevo proceso para ejecutar un programa. Ocurre por las relaciones de creación de procesos.
- **Nuevo → Listo:** El SO mueve a un proceso de estado nuevo a listo cuando éste se encuentre preparado para añadir un proceso en cuestión de recursos.
- **Listo → Ejecutando:** El SO selecciona uno de los procesos que se encuentre en el estado Listo. Esto lo hace el planificador a corto plazo.
- **Ejecutando → Saliente:** El proceso actual en ejecución se finaliza por parte del SO por cualquiera de las causas de salida o terminación.
- **Ejecutando → Listo:** El proceso en ejecución alcanza el mayor tiempo posible ejecutando ininterrumpidamente (alarma de reloj). Hay otras causas:
 - Atender a una interrupción de Entrada/Salida.
 - Que un proceso A está ejecutando a un nivel de prioridad, y uno B que se encuentra Bloqueado tiene un nivel de prioridad mayor.
 - En una política basada en prioridades con determinadas excepciones, el planificador puede introducir en CPU un proceso distinto del A actual, que puede tener menos prioridad, y provocaría un evento que liberaría a otro B Bloqueado con mayor prioridad que A.
- **Ejecutando → Bloqueado:** Si solicita algo por lo que deba esperar:
 - Operación de E/S.
 - Llamada bloqueante (wait()).
 - Cuando un proceso se comunica con otro, uno se bloquea mientras espera que otro le pase datos como usar colas de mensajes.
- **Bloqueado → Listo:** Cuando sucede el evento por el que estaba esperando.
- **Listo → Saliente:** Un padre puede terminar la ejecución de un proceso hijo en cualquier momento. Se da si un proceso es terminado mediante una señal enviada por parte de otro proceso (kill).
- **Bloqueado → Saliente:** Ocurre lo mismo que en el caso anterior.

6.2 UN POSIBLE ESQUEMA DE COLAS EN RELACIÓN A LOS ESTADOS

A continuación, se muestra un esquema de dos colas: Listos y Bloqueados:



Cada proceso admitido por el sistema se coloca en la cola de Listos. El SO selecciona uno de la cola de Listos que puede ser una lista de tipo FIFO. Cuando el proceso en ejecución termina de utilizar el procesador: o finaliza, o se coloca en la cola de Listos o Bloqueados, dependiendo de las circunstancias.

Cuando sucede un evento, cualquier proceso en la cola de Bloqueados que únicamente esté esperando a dicho evento, se mueve a la cola de Listos. El SO debe recorrer la cola entera de Bloqueados, buscando los procesos que estén esperando por dicho evento. Esto puede significar cientos o miles de procesos en esta lista por lo que sería más eficiente tener una cola por cada evento. Un refinamiento final sería: tener varias colas de procesos Listos, una por cada nivel de prioridad.

6.3 PROCESOS SUSPENDIDOS

Hay otros 2 estados como son el Listo/Suspendido y el Bloqueado/Suspendido. Cada proceso que se ejecuta en un sistema debe cargarse completamente en memoria principal. Esto puede no ser posible debido a:

- Tener que dejar libre memoria RAM para un proceso con más prioridad.
- La RAM está llena y no caben más procesos.

Lo que se hace es pasar procesos que estén en Listo o Bloqueado a swap (disco duro) por los estados Listo/Suspendido y Bloqueado/Suspendido. Cuando el proceso está Suspendido no está listo para su ejecución, ya que sólo está en memoria principal su BCP, al estar el resto en la memoria virtual, el sistema se volverá más lento:

- **Listo/Suspendido:** El proceso está en almacenamiento secundario, pero está disponible para su ejecución cuando sea cargado en memoria principal.
- **Bloqueado/Suspendido:** El proceso está en almacenamiento secundario y esperando un evento.

Cuando el SO realiza alguno, hay 2 opciones para seleccionar un nuevo proceso:

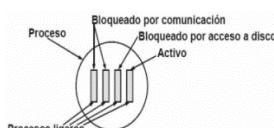
- Puede admitir un Nuevo proceso creado por el planificador a largo plazo.
- Puede traer un proceso que anteriormente se encontrase en Suspendido.

7. HILOS (HEBRAS)

Un hilo, también llamado proceso ligero o hebra, es una unidad básica de utilización de la CPU. Si un proceso tradicional (proceso pesado) tiene múltiples hilos, podría hacer más de una tarea a la vez si el sistema tiene varios núcleos.

Cada hilo perteneciente a un proceso posee lo siguiente de manera individual:

- Un estado de ejecución por hilo.
- Un contexto o BCP de hilo propio.
- Una pila de ejecución por hilo con sus registros de activación.



Cada hilo posee información que es compartida con el proceso al que pertenece:

- **Zona de código:** Los hilos se crean desde el código del proceso padre.
- **Zona de datos estáticos:** Las variables globales hay que protegerlas adecuadamente para no corromperlas (semáforos).
- **Montículo:** La reserva de memoria que haga un proceso padre o un hijo es visible para el resto de los hilos.
- **Otros recursos asociados al proceso:** Señales y manejadores de señales, directorio de trabajo actual, variables de entorno del proceso.

Todos los hilos residen en el mismo espacio de direcciones en RAM y tienen acceso a los mismos datos reservados en el montículo. Hay que tener cuidado cuando desde una hebra se quiere devolver información, ya que si no la ha reservado en el montículo ésta se perderá al terminar el hilo. Cuando un hilo cambia datos en memoria de montículo, otros hilos ven los resultados cuando acceden a estos datos.

7.1 ESTADO DE LOS HILOS

Los principales estados de los hilos son: Ejecutando, Listo y Bloqueado.

Suspender un proceso implica expulsar el espacio de direcciones de un proceso de memoria principal para dejar hueco a otro espacio. Todos los hilos comparten el mismo espacio de direcciones y se suspenderían al mismo tiempo. La finalización de un proceso termina todos los hilos de este ya que están dentro del mismo espacio.

7.2 MOTIVACIÓN Y VENTAJAS DE LAS HEBRAS

Una aplicación se implementa como un proceso propio con varias hebras de control.

- **Servidor web:** El servidor funciona como un solo proceso de aceptación de solicitudes. Cuando el servidor recibe una solicitud, crea otro hilo para dar servicio a dicha solicitud. Si fuera así, sólo podría atender a una solicitud y después otra, y no todas a la vez y los clientes tendrían que esperar.
- **Para trabajos en primer plano y en segundo plano:** Un procesador de textos tiene una hebra para mostrar gráficos, otra para pulsaciones de teclado del usuario y otra para el corrector.

Las ventajas de la programación multihebra se divide en 4 categorías:

1. **Utilización sobre arquitecturas multiprocesador:** Las hebras pueden ejecutarse en paralelo en los diferentes procesadores.
2. **Capacidad de respuesta o disponibilidad:** Permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado.
3. **Economía y compartición de recursos:** Las hebras comparten parte de la memoria y los recursos del proceso al que pertenecen, evitando la duplicidad.
4. **Tiempo de ejecución:**
 - Crear o terminar una hebra es mucho más rápido porque las hebras comparten recursos del padre que no hay que crear de nuevo.
 - El cambio de contexto entre hebras es más rápido que entre procesos ya que los hilos al pertenecer al mismo proceso, al hacer un cambio, el tiempo es casi despreciable (hay estructuras que no hay que salvar).
 - Comunicación entre hebras también más rápido ya que los datos están inmediatamente habilitados y disponibles entre hebras.

Si esta comunicación es entre hilos de distintos procesos debe intervenir el núcleo, en caso contrario no haría falta.

Por tanto, a la hora de implementar una aplicación es mucho más eficiente hacerlo con un conjunto de hilos que con un conjunto de procesos independientes.

7.3 ***MODELO MULTIHILO UNO A UNO***

Asigna cada hebra de usuario a una del kernel. Permite que se ejecute otra hebra mientras una hebra hace una llamada bloqueante. Permite que se ejecuten múltiples hebras en paralelo sobre varios procesadores. La planificación realizada por el núcleo se realiza a nivel de hilo.

El inconveniente de este modelo es que la gestión de cada hebra (crearla...) requiere una llamada al kernel por el uso de las correspondientes llamadas al sistema, produciendo un cambio a modo núcleo. Esto supone una carga de trabajo administrativa adicional, por lo que la mayoría de las implementaciones restringen el número de hebras soportadas por el sistema.

8. SERVICIOS EN GNU/LINUX

Un demonio o servicio es un proceso que se ejecuta en segundo plano, fuera del control directo de los usuarios. Esperan a que ocurran eventos y ofrecen servicios. Tradicionalmente los nombres de los demonios terminaban por la letra “d”.

El sistema inicia los demonios durante el arranque, aunque se pueden lanzar nuevos demonios en cualquier momento. Algunos de los más usuales son:

- **acpid:** Maneja el apagado del sistema cuando se pulsa el botón de encendido del equipo, y controla la energía a llevar a cabo al pulsar ciertos botones.
- **cron:** Para lanzar tareas planificadas.
- **networking:** Necesario para que funcione la interfaz de red.
- **apache2:** Permite servir páginas Web. Está siempre esperando solicitud.
- **smbd:** Para poder acceder a carpetas, archivos e impresoras de red (Samba).
- **cups:** Servicio de impresión, enviar y recibir información de la impresora.
- **udevd:** Detección de hardware.
- **alsa-utils:** Necesario para poder activar y controlar el sistema de sonido.
- **udisksd:** Para montar en el arranque determinados sistemas de ficheros.
- **lm-sensors:** Para hacer auditorias y sensores (frecuencia de trabajo de CPU).
- **bluetooth:** Encargado de las conexiones por Bluetooth e infrarrojos.
- **Servidor de correo:** Para poder aceptar conexiones de correo electrónico.

8.1 ***REGISTRO DE LA ACTIVIDAD DE LOS DEMONIOS***

Un demonio no debe estar conectado a ninguna terminal o Shell, ya que cualquier proceso que se execute desde el Shell es hijo de este, y si se cierra la terminal, se cerraría el demonio. Las condiciones de error las indica:

- Algunos demonios almacenan los mensajes en archivos específicos.
- Existe un demonio al que otros demonios pueden enviar mensajes para registrar sus eventos.

La mayoría de las distribuciones Linux guardan los archivos específicos en el directorio **/var/log** donde algunos están protegidos. Suelen tener los siguientes archivos:

- **syslog**: Registro del sistema de registro.
- **auth.log**: Contiene información de autorización del sistema (inicio de sesión).
- **cron.log**: Registra tareas de cron. Puede estar deshabilitado y escribir syslog.
- **daemon.log**: Registra alertas de servicios como ntfs-3g (L/S). Puede estar deshabilitado, en blanco y escribir syslog.
- **dmesg**: Se almacena la información que genera el kernel durante el arranque del sistema (información sobre los dispositivos que el kernel detectó).

8.2 ARRANQUE DE SERVICIOS CON UPSTART (BASADO EN SYSVINIT)

Los demonios se han lanzado clásicamente a través del proceso init. Upstart es un sistema de arranque basado en SysVinit. Systemd está sustituyendo a Upstart y es el que se utiliza actualmente y en algunos sistemas convive con Upstart.

El proceso init es llamado por el núcleo para iniciar el espacio de usuario durante el proceso de arranque y gestionar posteriormente todos los demás servicios y procesos a nivel de usuario. Por tanto, init es el primer proceso que se inicia y una de sus funciones es lanzar los demonios necesarios a partir de scripts.

init opera indicando un nivel de ejecución estableciendo, entre otras cosas, qué demonios deben ser iniciados a través del lanzamiento. Existen 7 niveles (del 0 al 6):

- Cuando la computadora entra al runlevel 0 está apagada.
- Cuando entra al runlevel 6 se reinicia.
- Los runlevels intermedios difieren en relación con qué servicios son iniciados.

Los scripts de bash que puede lanzar init se encuentran en el directorio /etc/init.d/. Estos se encuentran enlazados mediante enlaces simbólicos desde /etc/rc0.d/ a /etc/rc6.d/, uno para cada nivel.

Los niveles más bajos en Linux se usan para el mantenimiento o recuperación de emergencia:

Nivel de ejecución	Nombre o denominación	Descripción
0	Alto	Alto o cierre del sistema (apagado). Se ejecutan scripts para finalizar demonios. No se debe poner este nivel como predeterminado.
1	Modo monousuario	No configura la interfaz de red o los demonios de inicio, ni permite que ingresen otros usuarios que no sean el usuario root. Este nivel de ejecución permite reparar problemas, o hacer pruebas en el sistema.
2	Multiusuario	Multiusuario sin soporte de red (sin NFS o xinetd - demonio de servicios extendidos de Internet-)
3	Multiusuario con soporte de red	Inicia el sistema normalmente. GNU/Linux completamente funcional con soporte multiusuario y acceso a la red. La interfaz de usuario es en modo texto.
4	Multiusuario con soporte de red.	Similar al 3 o sin uso, reservado para administradores de sistemas.
5	Multiusuario gráfico (X11)	Nivel de ejecución 3 + display manager (entorno gráfico usando gestores de pantalla como gdm - GNOME Display Manager-). Normalmente siempre se arranca en este nivel.
6	Reinicio	Se reinicia el sistema. Se ejecutan scripts para finalizar demonios. No se debe poner este nivel como predeterminado.

Durante el arranque del sistema se verifica si existe un nivel de ejecución rc predeterminado. Después se ejecutan todos los enlaces simbólicos /etc/rc?.d relativos al nivel de ejecución. Después el proceso init se “alearga” y espera que suceda alguno de estos:

- Que procesos comenzados terminen o mueran y los recoge como proceso padre.
- Una petición a través del proceso /sbin/telinit para cambiar nivel de ejecución.

En el apagado o en el reinicio de la computadora, init es llamado a cerrar todas las funcionalidades del espacio de usuario a través de scripts tras lo que init termina y el núcleo ejecuta el apagado. A estos scripts se le pone como prefijo “K” de “Kill”.

Al usar telinit para cambiar el nivel de ejecución, de dejará de tener una pantalla gráfica y aparecerá una terminal, y es probable que se pierda la información no guardada. Este comando no altera el contenido de los archivos de configuración de más alto nivel. Para ver el nivel de ejecución actual se usa runlevel y who -r.

```
jfcaballero@eniac: ~
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:~$ runlevel
N 5
jfcaballero@eniac:~$ who -r
`run-level' 5 2021-11-10 08:54
jfcaballero@eniac:~$
```

Los parámetros que admiten un determinado servicio se pasan mediante el comando service y el nombre del servicio (# service bluetooth). Algunos comandos son:

- # service bluetooth start: Inicia el servicio indicado.
- # service bluetooth stop: Para el servicio indicado.
- # service bluetooth restart: Reinicia el servicio indicado.
- # service bluetooth force-reload: Recarga el servicio indicado sin llegar a pararlo. Si el servicio no permite la recarga, se reinicia.
- # service bluetooth status: Muestra información detallada sobre su estado.
- # update-rc.d: Sirve para automatizar el proceso de enlazado simbólico, según en los niveles que quiera incluir un servicio.

8.3 ARRANQUE DE SERVICIOS CON EL SISTEMA SYSTEMD

El principal motivo de la aparición de systemd es unificar las configuraciones básicas de la administración de servicios y su conexión con el núcleo. Systemd integra más partes del sistema, es un “super servicio”. Además:

- Permite en el arranque la ejecución paralela de procesos (init no lo permite).
- Reemplaza varios servicios del sistema. journal es el sistema de registros y reemplaza o convive con syslog.

Genera controversia en los siguientes aspectos:

- Arquitectura interna: Se critica que es demasiado complejo y más complicado de administrar que Upstart.
- Implementación futura: Que se torne en un sistema de dependencias entrelazadas, obligando a los mantenedores de distribuciones a depender de systemd ya que cada vez más piezas de software del espacio de usuario dependen de sus componentes.

Los demonios se definen y configuran en los llamados archivos de unidad o unit files que son ficheros con un tipo de extensión definida y que se alojan en varios directorios. En `systemd` no existen diferentes niveles de arranque en los subdirectorios que usaba `init`, **se ha reemplazado por los target**, que son **ficheros de unidad con extensión .target de los que dependen los ficheros .service**.

Algunas equivalencias entre Upstart y Systemd son:

Nivel Upstart	Unidades .target	Descripción
0	<code>runlevel0.target</code> , <code>poweroff.target</code>	Alto o cierre del sistema (apagado), mediante el comando <code>halt</code> . No se debe poner este nivel como predeterminado.
1	<code>runlevel1.target</code> , <code>rescue.target</code>	No configura la interfaz de red o los demonios de inicio, ni permite que ingresen otros usuarios que no sean el usuario <code>root</code> . Este nivel de ejecución permite reparar problemas, o hacer pruebas en el sistema.
2	<code>runlevel2.target</code> , <code>multi-user.target</code>	Multiusuario sin soporte de red (sin NFS o xinetd - demonio de servicios extendidos de Internet-)
3	<code>runlevel3.target</code> , <code>multi-user.target</code>	Inicia el sistema normalmente. GNU/Linux completamente funcional con soporte multiusuario y acceso a la red. La interfaz de usuario es en modo texto.
4	<code>runlevel4.target</code> , <code>multi-user.target</code>	Similar al 3 o sin uso, reservado para personalización para administradores de sistemas.
5	<code>runlevel5.target</code> , <code>graphical.target</code>	Nivel de ejecución 3 + display manager (entorno gráfico usando gestores de pantalla como <code>gdm</code> - <i>GNOME Display Manager</i> -). Normalmente siempre se arranca en este nivel.
6	<code>runlevel6.target</code> , <code>reboot.target</code>	Se reinicia el sistema. No se debe poner este nivel como predeterminado.

De los 12 tipos, las más interesantes son service y target:

- **Service:** Controla demonios y los procesos relativos a ellos. Interesa para crear un nuevo servicio.
- **Target:** Codifica información respecto a agrupar unidades .service y lograr una buena sincronización entre ellas. Se encargan de las dependencias entre unidades y relaciones entre sí.

8.3.1 Systemctl (el sustituto de service)

El comando `service` de `init` se sustituye o convive con el comando `systemctl` de `systemd`. Para activar demonios permanentemente en la secuencia de arranque hay que ejecutar:

```
#systemctl enable unit_name.service
```

Siendo `unit_name` el nombre del servicio o unidad. Al ejecutarlo se creará un enlace simbólico en el directorio `/etc/systemd/system`. Para eliminar el servicio:

```
#systemctl disable unit_name.service
```

Para iniciar, reiniciar, recargar o parar un servicio se usan los siguientes comandos:

- **# sudo systemctl start unit:name.service:** Para iniciar un servicio.
- **# sudo systemctl restart unit:name.service:** Para reiniciar un servicio.
- **# sudo systemctl reload unit:name.service:** Para recargar un servicio sin llegar a pararlo o reiniciarlo.
- **# sudo systemctl stop unit:name.service:** Para detener un servicio.

Otros comandos que pueden ser útiles son:

- **# sudo systemctl set-default unit:name.target:** Establece el nivel .target por defecto en el arranque del sistema.
- **# systemctl list-units --type target:** Indica los .target cargados en el sistema que implicará cargar una serie de unidades tipo .service.
- **# systemctl list-dependencies default.target:** Consultar las dependencias del arranque por defecto.
- **# systemctl list-units --type service:** Permite obtener la totalidad de servicios activos:
 - UNIT: El nombre de la unidad.
 - LOAD: Si la información está cargada en memoria.
 - ACTIVE: Informa si la unidad está activa o no.
 - SUB: Muestra más información sobre la unidad.
 - DESCRIPTION: Una descripción breve de lo que hace la unidad.
- **systemctl list-units --type service -all:** Listado de la totalidad de servicios independientemente de si están activos o no.
- **Systemctl status unit_name.service:** Permite ver información de estado sobre el servicio:
 - Loaded: Sobre si la unidad ha sido cargada y la ruta absoluta.
 - Active: Sobre si la unidad de servicio se está ejecutando.
 - Main PID: El PID del servicio, seguido de su nombre.
 - Process: Información adicional sobre procesos relacionados.
 - CGroub: Información adicional sobre si ese servicio pertenece a un determinado grupo de servicios.
- **journalctl:** Permite ver el registro con la información que genera el núcleo en el arranque del sistema (similar a dmesg de Upstart).
- **journal -u unit_name.service:** Permite ver el registro generado por el servicio.

TEMA 3: COMUNICACIÓN ENTRE PROCESOS

1. COMUNICACIÓN ENTRE PROCESOS, PROBLEMAS DE CONCURRENCIA

Dos procesos se dice que son concurrentes si hay una existencia simultánea de varios procesos en ejecución, aunque esta no implique una ejecución simultánea. El término concurrencia da lugar a dos tipos de paralelismo respecto a la ejecución de procesos:

- **Pseudoparalelismo o concurrencia:** Los procesos se intercalan en el tiempo en un solo procesador (un único núcleo).
- **Paralelismo o concurrencia real:** Los procesos se ejecutan de manera paralela en el mismo instante de tiempo, cada uno en un procesador diferente o diferentes núcleos.

Aunque parece que la intercalación o “multiprogramación” y la superposición o “multiprocesamiento” de la ejecución de procesos presentan formas de ejecución distintas, ^{PERO} se verá que presentan los mismos problemas cuando quieren hacer uso de ^{AL} recursos compartidos.

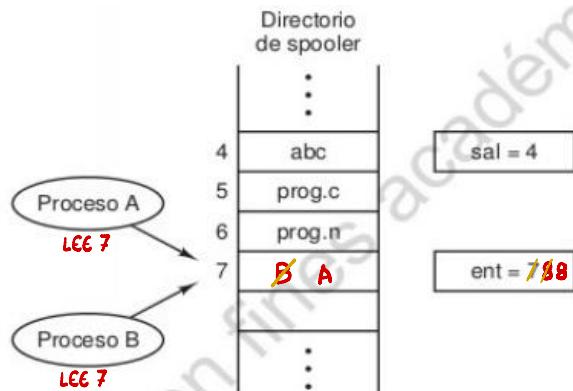
1.1 EJEMPLO

Consideremos un spooler o cola de impresión. Un proceso introduce el nombre del archivo en un directorio de spooler especial. Otro proceso comprueba en forma periódica si hay archivos que deban imprimirse y los imprime y borra sus nombres del directorio.

Suponiendo que el spooler tiene una cantidad muy grande de ranuras, cada una capaz de contener el nombre de un archivo. Hay 2 variables:

- sal: Apunta al siguiente archivo a imprimir.
- ent: Es una variable compartida que apunta a la siguiente ranura libre.

Suponiendo que las ranuras de la 0 a la 3 están vacías y las ranuras de la 4 a la 6 están llenas, los procesos A y B deciden que desean poner en cola un archivo para imprimirllo. Podría ocurrir lo siguiente:



- El proceso A lee ent y guarda el valor 7 en una variable local.
- Ocurre una interrupción de reloj y la CPU decide que el proceso A se ha ejecutado el tiempo suficiente y conmuta al proceso B.
- El proceso B también lee ent y también obtiene un 7 y lo almacena en su variable local y ambos piensan que la siguiente ranura libre es la 7.
- El proceso B continua su ejecución. Almacena el nombre de su archivo en la 7 y actualiza ent a 8.
- El proceso A se ejecuta de nuevo, partiendo de donde se quedó. Busca en su variable, encuentra un 7 y escribe el nombre de su archivo borrando el nombre que el proceso B había puesto y actualiza ent a 8.

El directorio de spooler es ahora internamente consistente, por lo que el demonio de impresión no detectará nada incorrecto, pero el proceso B nunca recibirá ninguna salida y el usuario B esperará por años y nunca llegará la salida.

1.2 CONCLUSIONES SOBRE LOS EJEMPLOS, CONDICIONES DE CARRERA

Es necesario proteger las variables y recursos compartidos y la única forma de hacerlo es controlar el código que accede a esos recursos. Las situaciones en las que dos o más procesos están leyendo o escribiendo datos compartidos y el resultado depende de quién se ejecuta y cuando lo hace, se llama condiciones de carrera.

2. SECCIÓN CRÍTICA Y EXCLUSIÓN MUTUA

Para evitar las condiciones de carrera hay que buscar alguna manera de prohibir que más de un proceso lea y escriba sobre dichos recursos al mismo tiempo. Esta parte del programa en la que se accede a datos compartidos se llama sección crítica. Lo que necesitamos es exclusión mutua (no más de un proceso ejecutando la sección crítica).

CRUCE CON ATASCO

3. INTERBLOQUEO E INANICIONES POR ENTRADA A LA SC

Utilizando los mecanismos de acceso en exclusión mutua se puede dar interbloqueo que es el bloqueo permanente de un conjunto de procesos que compiten por recursos del sistema o se comunican entre sí.

Un conjunto de procesos está interbloqueado cuando cada proceso está esperando un evento, normalmente la liberación de algún recurso requerido, que sólo puede generar otro proceso bloqueado del conjunto. Por ejemplo: un proceso A está bloqueado esperando por un recurso que tiene otro proceso B, que a su vez está bloqueado a la espera de un recurso que tiene el proceso A.

Existen algoritmos y técnicas para prevenir y detectar interbloqueos entre procesos, pero son relativamente complejas como dar “un paso atrás” para pasar al estado previo en el que se produjo la problemática e intentar salvar el interbloqueo.

Se debe evitar la inanición en cuanto a la imposibilidad para acceder a la sección crítica. Es similar al interbloqueo, pero se da inanición cuando uno o más procesos están esperando recursos ocupados por otros procesos, pero no se encuentran necesariamente en ningún punto muerto.

En la utilización de prioridades, podía ocurrir que procesos de alta prioridad estuvieran ejecutándose siempre y no permitieran la ejecución de procesos de baja prioridad, causando inanición en estos.

Para evitarlo, los planificadores incorporan algoritmos para asegurar que todos los procesos reciben un mínimo de tiempo de CPU. Hay mecanismos como los semáforos fuertes que no permiten que se produzca inanición por largos períodos de tiempo.

Un caso especial de inanición se conoce como inversión de prioridad, que es que, si un proceso de alta prioridad está pendiente del resultado o de un recurso de un proceso de baja prioridad que no se ejecuta nunca, entonces este proceso de alta prioridad también experimente inanición.

4. SOPORTE SOFTWARE PARA LA EXCLUSIÓN MUTUA

Todas las técnicas software pueden llegar a funcionar sólo si se asume:

- **Exclusión mutua elemental a nivel de acceso a memoria:** Se serializan accesos simultáneos a la misma ubicación de memoria principal por alguna clase de árbitro de memoria (controladora) de memoria y de acceso a los buses del sistema). Es un control hardware.
- **Los procesos involucrados no se caen del sistema mientras estén dentro de ella:** Si un proceso termina antes de salir de la sección crítica las técnicas software no funcionan, hay que implementar mecanismos más complejos para su control y evitar bloqueos.

En los SO actuales, a nivel de núcleo, se implementan mecanismos para que los procesos que están utilizando un recurso compartido del sistema y que terminan de manera abrupta cuando lo están usando, no lo bloquen permanentemente y éste sea liberado para que lo usen otros procesos.

4.1 ALGORITMO DE DEKKER

4.1.1 Tentativa primera

- Se reserva una ubicación de memoria compartida etiqueta como turno=0.
- Un proceso que desee ejecutar su sección crítica examina primero el contenido de turno.
- Si el valor de turno es igual al número del proceso, entonces puede acceder a su sección crítica.
- En caso contrario, si turno no es igual al número, está forzado a esperar.
- Despues de que haya obtenido el acceso a su sección crítica y de completar dicha sección, debe actualizar el valor de turno para el resto de los procesos. Hay una variable compartida que controla el acceso a la sección crítica que se inicia en 0.

Esta solución garantiza la propiedad de exclusión mutua, pero tiene desventajas:

1. Los procesos deben alternarse estrictamente en el uso de su sección crítica, por lo que el ritmo de ejecución viene determinado por el proceso más lento.
2. Si un proceso cae antes de establecer el valor de turno, el otro se encuentra permanentemente bloqueado, en espera activa.

3. Hay espera activa en el bucle `while()`. El término espera activa o cíclica se refiere a una técnica en la que un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero sigue ejecutando una o varias instrucciones que comprueben la variable apropiada para conseguir entrar. La espera activa consume ciclos de reloj sin hacer nada.
4. El proceso que espera lee de forma repetida el valor de la variable turno hasta que puede entrar en la sección crítica. Esto es la espera activa (busy waiting).

4.1.2 Tentativa segunda

El problema de la primera es que cada proceso debería tener su propia llave para entrar en la sección crítica al ritmo que desee. Se realiza lo siguiente:

- Se define un vector compartido estado con `estado[0]` y `estado[1]` donde cada elemento puede tener 2 valores (0 o 1).
- Cada proceso puede examinar el estado del otro pero no alterarlo, sólo puede cambiar su propio valor.
- Cuando desea entrar en su sección crítica, periódicamente comprueba el estado del otro hasta que tenga el valor 0, el otro proceso no está en la crítica.
- El proceso que está realizando la comprobación inmediatamente establece su propio estado a 1 (va a entrar en la sección crítica).
- Cuando deja su sección crítica, establece su estado a 0.

Con esta tentativa, se resuelve la alternancia, un proceso puede entrar en su sección crítica tan frecuentemente como desee, pero sigue habiendo desventajas:

- No se garantiza la exclusión mutua ya que ambos procesos estarían en condiciones de entrar en la sección crítica.
- Si un proceso cae antes del código de establecimiento de estado a 0, el otro se queda bloqueado.
- Sigue habiendo espera activa.

4.1.3 Tentativa tercera

El problema del acceso a la vez a la sección crítica por parte de los 2 procesos se puede arreglar con un simple intercambio de 2 sentencias poniendo `estado[i]` antes del `while()`. Así se garantiza la exclusión mutua, pero se genera otro problema:

- Si ambos procesos establecen su estado a 1 antes de que se haya ejecutado la sentencia `while`, los dos pensarán que está ocupada y causa un interbloqueo.
- Si un proceso cae antes del código de establecimiento de estado a 0, el otro se queda bloqueado.
- Sigue habiendo espera activa.

4.2 ALGORITMO DE PETERSON

Hay una variable compartida `estado` que indica la posición de cada proceso con respecto a la exclusión mutua:

- Se define un vector compartido estado con `estado[0]` y `estado[1]`.
- Una variable compartida turno que resuelve conflictos simultáneos.

Tiene las siguientes ventajas:

- Garantiza la exclusión mutua.
- Evita interbloqueos en el bucle while con respecto a la solución anterior.
- Evita que los procesos puedan estar utilizando su sección crítica repetidamente.

La solución de Peterson sigue teniendo 2 problemas:

- Si un proceso cae antes del código de establecimiento de estado a 0, el otro se queda bloqueado.
- Sigue habiendo espera activa.

La espera activa se puede resolver con semáforos.

5. SOPORTE HARDWARE PARA LA EXCLUSIÓN MUTUA

5.1 INSTRUCCIÓN TEST AND SET (TSL)

O - O para
O - 1 Sigue
1 - O Pasa
1 - 1 Espera

Los diseñadores de procesadores han propuesto instrucciones máquina que se pueden invocar desde código ensamblador y que llevan a cabo dos acciones atómicamente, comprobar y escribir sobre una única posición de memoria con un único ciclo de búsqueda-ejecución de instrucción.

TEST y SET son las siglas de Test and Set Lock, utilizada para facilitar la creación de semáforos y otras herramientas necesarias para la programación concurrente en computadores. Realiza dos acciones atómicamente: comprobar el contenido de una palabra de la memoria y almacenar un valor distinto de 0 en la palabra de memoria.

La solución TSL seguirá teniendo los siguientes problemas:

- Hay espera activa.
- Si un proceso cae antes del código de establecimiento de cerrojo a 0, el resto quedan bloqueados.
- No hay orden de acceso a la sección crítica, entrará el que llegue antes a ejecutar la TSL. Esto no ocurre en los semáforos fuertes ya que el orden de ejecución es FIFO.

6. SEMÁFOROS BINARIOS SEMAPHORE

El tratamiento de los problemas de programación concurrente se denominan semáforos, que resuelven la espera activa de los algoritmos clásicos. El principio es:

Dos o más procesos pueden cooperar por medio de señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Para esta señalización se pueden utilizar unas variables especiales llamadas semáforos. A un semáforo binario se les conoce también como mutex, o barrera. O CANDADO

Estos semáforos también poseen una cola, una variable que puede tomar 2 valores y 2 primitivas atómicas. Solo puede tomar los valores 0 (cerrado) y 1 (abierto).

Se puede definir con las siguientes 3 instrucciones:

1. Se puede inicializar a un valor 0 (cerrado) o 1 (abierto).
2. La operación semWaitB() comprueba el valor del semáforo: si el valor es 1, se cambia a 0 y sigue ejecutándose. Si es 0, el proceso se bloquea.
3. La operación semSignalB() comprueba si hay algún proceso bloqueado:
 - o Si hay algún proceso bloqueado se desbloquea.
 - o Si no hay procesos bloqueados, el valor del semáforo se establece a 1.

Las acciones de comprobar el valor, modificarlo y pasar a bloqueado, se realizan en conjunto como una sola acción atómica indivisible (se encarga de ello el núcleo del SO). Es decir, no se permite que se produzcan interrupciones mientras se ejecutan esas 2 primitivas, al igual que ocurre con Tast&Set().

7. SEMÁFOROS GENERALES

Se describen los semáforos generales o semáforos con contador, que además de proteger una sección crítica pueden sincronizar procesos en función de determinadas condiciones. Se tiene que:

- Para recibir una señal vía el semáforo s, el proceso ejecutará la primitiva semWait(s).
- Para transmitir una señal, el proceso ejecutará la primitiva semSignal(s). Si no se ha transmitido aun, el proceso se bloquea hasta que ocurra.

El semáforo puede implementar una variable que contiene un valor entero, sobre el cual sólo están definidas 3 operaciones:

1. Un semáforo puede ser inicializado a un valor no negativo, normalmente a 1.
2. La operación semWait() decrementa el valor del semáforo (s.cuenta--). Si el valor pasa a ser <0, el proceso se bloquea, sino continua la ejecución.
3. La operación semSignal() incrementa el valor del semáforo (s.cuenta++). Si el valor es <=0, se desbloquea uno de los procesos bloqueados.

Una vez que empieza una operación semWait() o semSignal(), ningún otro proceso podrá acceder al semáforo sino hasta que la operación se haya completado al ser operaciones atómicas.

- Cada proceso ejecuta un semWait(s) justo antes de entrar en su sección crítica.
- El primer proceso que ejecute el semWait() será capaz de entrar en su sección crítica inmediatamente poniendo s a 0.
- Cualquier otro proceso que intente entrar en su sección crítica la encontrará ocupada y se bloqueará, poniendo s=-1.
- Cada intento insatisfactorio conllevará otro decremento del valor de s.
- Cuando el proceso inicialmente entró en su sección crítica salga de ella, s se incrementa y uno de los procesos bloqueados se extrae de la lista de procesos bloqueados asociada con el semáforo y se pone en estado Listo y entra en la sección crítica.

El orden en que los procesos deben ser extraídos de la cola:

- La más favorable es FIFO (el primero que entra es el primero que sale). El proceso que lleva más tiempo bloqueado es el primero en ser extraído. Este semáforo se llama semáforo fuerte, los cuales garantizan estar libres de inanición.
- Un semáforo que no especifica el orden es un semáforo débil.

Si se necesita que se permita más de un proceso en su sección crítica a la vez, casos muy específicos hay que iniciar el semáforo al valor correspondiente al número de procesos en lugar de que sea 1.

8. PROBLEMA DE LOS LECTORES-ESCRITORES

- Hay un objeto de datos compartido que es utilizado por varios procesos, unos que leen y otros que escriben.
- Sólo puede utilizar el objeto recurso por un proceso y sólo uno, es decir, o bien un proceso estará escribiendo o bien leyendo, pero nunca simultáneamente.

La solución a este problema sería implementar un algoritmo con semáforos para que:

- Cualquier número de lectores puedan leer del fichero simultáneamente.
- Sólo un escritor al tiempo puede escribir en el fichero.
- Si un escritor está escribiendo, ningún lector puede leerlo, no se puede escribir y leer a la vez.

Si queremos darles prioridad a los lectores, es decir, si hay lectores leyendo pueden seguir entrando en la sección crítica más lectores, aunque ya haya un escritor bloqueado porque haya intentado acceder antes que estos nuevos lectores en el tiempo a esta sección crítica.

```
# Semáforos y variables compartidas
mutex = Semaforo(1)          # Para proteger el acceso a las variables compartidas
writerLock = Semaforo(1)       # Para bloquear a los escritores cuando haya lectores
readerCounter = 0             # Número de lectores activos

# Proceso de lector
Lector:
    wait(mutex)              # Protege el contador de lectores
    readerCounter += 1        # Incrementa el número de lectores activos
    if readerCounter == 1:
        wait(writerLock)     # El primer lector bloquea a los escritores
        signal(mutex)         # Libera el acceso al contador de lectores

# Sección crítica para lectura
Leer()                      # Realiza la operación de lectura

    wait(mutex)              # Protege el contador de lectores
    readerCounter -= 1        # Decrementa el número de lectores activos
    if readerCounter == 0:
        signal(writerLock)   # El último lector libera el bloqueo para escritores
        signal(mutex)         # Libera el acceso al contador de lectores

# Proceso de escritor
Escritor:
    wait(writerLock)         # Espera hasta que no haya lectores activos
    # Sección crítica para escritura
    Escribir()               # Realiza la operación de escritura
    signal(writerLock)        # Libera el bloqueo para permitir a los lectores o a otros escritores
```

El proceso de resolución obtenido con el pseudocódigo anterior es el siguiente:

- El semáforo sc se utiliza para cumplir la exclusión mutua. Mientras un bibliotecario esté accediendo al área de datos compartidos, ningún otro bibliotecario ni lector puede acceder.
- La variable compartida cuentalect se utiliza para llevar la cuenta del número de lectores.
- El semáforo x se usa para asegurar que cuentalect se actualiza.
- Para permitir múltiples lectores, cuando no hay lectores leyendo, el primer lector que intenta leer debe llamar a semWait() en sc.
- Cuando ya haya al menos uno, los siguientes lectores no necesitan hacer dicha llamada y pueden entrar en la sección crítica de manera directa.
- Cuando el último lector salga de la sección crítica, la libera para que entre a un escritor o nuevos lectores.

9. PROBLEMA DEL PRODUCTOR-CONSUMIDOR

Es conocido como el problema del almacenamiento limitado o acotado. El problema:

- Hay un proceso productor generando algún tipo de datos y poniéndolos en un buffer.
- Hay un proceso consumidor que está extrayendo datos del buffer uno a uno.
- El sistema está obligado a impedir la superposición de las operaciones sobre los datos, sólo un agente puede acceder al buffer en un momento dado. El productor no sobrescribe un elemento que esté leyendo el consumidor, y viceversa.
- **CASO EXTREMO 1:** Si el buffer está lleno, el productor debe esperar hasta que el consumidor lea al menos un elemento para seguir almacenando datos.
- **CASO EXTREMO 2:** Si el buffer está vacío, el consumidor debe esperar a que se escriba información nueva por parte del productor.

Se necesita 3 semáforos, cuyas funciones son:

- **Semáforo sc**: Para cumplir la exclusión mutua en el acceso al buffer, iniciado a 1.
- **Semáforo cn**: Sirve para indicar el número de espacios ocupados en el buffer por la información que no se ha consumido. Se inicializa a 0. Controla que el consumidor se bloquee cuando no hay nada que consumir (cn.cuenta=0).
- **Semáforo pr**: Cuenta el número de espacios vacíos o libres del buffer. Se inicializa a N, siendo N el tamaño del buffer. Controla que el productor se bloquee si no tiene elementos o espacios libres donde almacenar información, es decir, si pr.cuenta=0.

```

buffin=0
buffout=0
semaforo s = 1; // Semáforo general para la sección crítica.
semaforo cn = 0; // Semáforo general para el consumidor.
semaforo pr = TAM_BUFFER // Semáforo general para el productor.

# Proceso productor
Productor:
    wait(pr); //Un espacio libre menos. Si pr.cuenta<0 me bloqueo.
    wait(s);
    buffer(buffin)=produce(); //Pone en el buffer un valor generado.
    buffin=(buffin+1)%TAM_BUFFER;
    signal(s);
    signal(cn); //Un espacio más ocupado por un dato generado.

# Proceso consumidor
Consumidor:
    wait(cn); //Un espacio menos ocupado, si cn.valor < 0 me bloqueo.
    wait(s);
    consume(buffer[buffout]); //Lee del buffer un valor.
    buffout=(buffout+1)%TAM_BUFFER;
    signal(s);
    signal(pr);

```

En el código del productor se observa:

- Se espera que haya elementos libres en el buffer, es decir, que el semáforo pr sea no nulo. El productor puede poner información en el buffer.
- Se comprueba si hay algún proceso que se encuentre en la sección crítica. El productor se quedará esperando, sino lo hay entrará en la sección crítica y pone la información producida en el buffer.
- Incrementar el semáforo cn, lo que indica que se ha introducido un nuevo elemento en el buffer.

En el código del consumidor se observa:

- Se espera a que haya elementos ocupados en el buffer, es decir, a que cn tenga un valor no nulo.
- Se realiza la comprobación sobre s para evitar problemas de exclusión mutua.
- Tras entrar en la sección crítica, se actualiza el semáforo pr.

Se utilizan los semáforos con 2 objetivos diferentes, EXCLUSIÓN y SINCRONIZACIÓN.

- El semáforo general s se usa para garantizar la exclusión mutua en la sección crítica.
- Los semáforos generales, pr y cn se utilizan como mecanismo de sincronización. Se aprovecha las características de los semáforos de ejecución invisible y ausencia de espera improductiva.

10. SEÑALES COMO MÉTODO IPC (INTERPROCESS COMMUNICATION)

Las señales son otra manera de crear comunicaciones entre procesos, por tanto, se encuentran dentro de los diferentes mecanismos de la comunicación inter-proceso.

Son un mecanismo de comunicación rápido y unidireccional por el que se envía un número mediante interrupciones de tipo software.

El número y tipo de señales viene impuesto por el SO y cada una de ellas será empleada en un caso concreto, siendo su número identificador la única información que realmente se transmite entre los procesos, y el significado depende de:

- La interpretación del programador (SIGUSR1 y SIGUSR2).
- Las establecidas por defecto en el sistema y que se interpretan y tratan de manera predeterminada por el núcleo.

Las señales son producidas:

- Directamente por el kernel y tiene como finalidad parar o desviar el curso normal de las instrucciones que se ejecutan en un determinado proceso.
- Un proceso puede enviar una señal a otro proceso a través de una llamada wrapper al sistema que desemboca en un cambio a modo núcleo.

Se pueden agrupar de la siguiente manera:

- Señales relacionadas con la terminación de procesos (SIGKILL).
- Señales relacionadas con las excepciones inducidas por los procesos (SIGBUS).
- Señales relacionadas con los errores irrecuperables originados en el transcurso de una llamada al sistema (SIGSYS).
- Señales propias originadas desde un proceso y definidas por el programador (SIGUSR1).
- Señales relacionadas con la intención con el terminal (SIGINT).
- Señales relacionadas con problemas de hardware (SIGIOT).

Cuando un proceso en ejecución recibe una señal tiene que interrumpirse, se carga la rutina ISR que la trata de 3 formas diferentes, y se salva su contexto:

1. Ignorar la señal mediante la asociación de su recepción con un tratamiento que no tiene efecto (ni tan siquiera su comportamiento por defecto).
2. Invocar a una rutina sin ese parámetro específico, la cual establezca en su cuerpo un tratamiento personalizado de la señal, ignorando su comportamiento por defecto. Se les conoce como callbacks o retrollamadas.
3. Realizar el tratamiento por defecto correspondiente al número de señal. Tiene como fin el terminar el proceso que recibe la señal.

10.1 SEÑALES EN GNU/LINUX

- **SIGHUP – Hangup:** Se envía a los procesos de un grupo cuando su líder de grupo termina su ejecución. Termina la ejecución del proceso que la recibe.
- **SIGINT – Interrupción:** Se envía cuando en una terminal en medio de un proceso se pulsan las teclas de interrupción. Termina la ejecución del proceso.
- **SIGFPE – Error en coma flotante:** Se envía cuando el hardware detecta un error en coma flotante. Termina el proceso que la recibe.
- **SIGKILL – Kill:** Provoca irremediablemente la terminación del proceso.
- **SIGSEGV – Violación de segmento:** Se envía a un proceso cuando intenta acceder a datos que se encuentran fuerza de su segmento de datos. Termina el proceso.

- **SIGALRM – Alarm clock:** Cada proceso tiene asignados un conjunto de temporizadores. Si se ha activado y llega a 0, se envía esta señal al proceso y continúa por la siguiente línea de código.
- **SIGTERM – Finalización software controlada:** Se envía para indicarle a un proceso que debe terminar su ejecución. Puede ser ignorada, pero por defecto termina el proceso.
- **SIGUSR1 – Señal 1 del usuario:** Está reservada para el usuario. Ninguna aplicación estándar va a utilizarla.
- **SIGUSR2 – Señal 2 del usuario:** Idéntico al SIGUSR1.

10.2 ENVÍO DE SEÑALES

Un proceso puede enviar señales a otro proceso mediante la función `kill()` que admite 2 parámetros:

```
#include <signal.h>
```

```
void kill(pid_t pid, int sig)
```

- `pid_t` es el identificador del proceso al que queremos enviar la señal. Si se pone un número mayor de 0, se envía al proceso cuyo ID de proceso coincide con el número.
- `sig` es el número o MACRO de la señal que queremos enviar.

Desde un terminal podemos enviar una señal a un proceso mediante el comando `kill`, seguido de la macro correspondiente a la señal.

La llamada al sistema `kill()`, ya sea a través de su programación o invocada desde un terminal, llega al núcleo a través de `system_call`, invocando `sys_call`.

10.3 ASIGNACIÓN Y RECEPCIÓN DE SEÑALES

Para tratar las señales se usa `signal()`, se usa de la siguiente manera:

```
#include <signal.h>
```

```
void* signal(int sig, void (*func)(int))
```

- `sig` es un número entero con el identificador o macro de la señal.
- `func` indica cómo se manejará la señal, es la señal callback.
 - Si el valor es `SIG_DFL`, se usa el manejo por defecto para la señal.
 - Si es `SIG_IGN`, la señal será ignorada.
 - Si no, se apuntará a una función callback o manejador de señal que hay que implementar y que se llamará cuando al señal se active.

Para usar un manejador de señal:

```
void callback(int);
```

```
signal(SIGINT, callback);
```

Después de invocar un signal() por parte de un proceso en ejecución, si se recibe una señal ocurrirá:

1. Se interrumpe la ejecución del proceso.
2. Se ejecuta la función manejadora ISR.
3. Se salva el contexto del proceso y pasa a Listo.
4. Cuando el planificador lo pase a estado Ejecutando se comienza a ejecutar el proceso por la función callback().
5. Cuando termine se continua por la sentencia donde el proceso se interrumpió.

10.4 ALARMAS Y PAUSAS

Se puede generar una alarma con la función alarm(). Esta se invoca por un proceso para decirle al núcleo que le envíe la señal SIGALRM al cabo de un cierto tiempo, pasa como parámetro un tiempo en segundos, se inicia la cuenta atrás de esa duración y el proceso continúa su ejecución.

Cuando se recibe la señal alarm(), por defecto el proceso seguiría ejecutando tras pasar a Listo por la siguiente línea de código por la que iba.

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned seconds)
```

- El parámetro seconds es el número de segundos que transcurrirán antes de que se ejecute la señal por parte del núcleo. Un proceso sólo puede tener una petición de alarma pendiente, cada nueva petición anula la anterior. Si se invoca alarm() con el parámetro 0, se cancelan las alarmas pendientes.

Puede que a veces sea interesante parar la ejecución de un proceso hasta que se produzca una señal. Para ello se puede usar la llamada pause().

```
#include <unistd.h>
```

```
int pause(void)
```

- Esta llamada hace que el proceso invocador se bloquee (quede en espera) hasta la llegada de una señal cualquiera.
- Después de ejecutarse devuelve el valor -1 y la ejecución se continúa con la sentencia que sigue a esta llamada.

TEMA 4: PLANIFICACIÓN

1. OBJETIVOS DE LA PLANIFICACIÓN

Los objetivos más relevantes son:

- **Justicia:** El algoritmo de planificación debe dar una porción de tiempo de CPU justa a todos los procesos, ningún proceso debe de acaparar la CPU.
- **Aplazamiento indefinido:** Se debe evitar la inanición de procesos que ocurre cuando un proceso no recibe nunca servicio del procesador.
- **Productividad:** Es la cantidad de trabajo finalizada por unidad de tiempo. Depende de la longitud de los procesos. Puede sobrecargar por el costo extra.
- **Costo extra:** Es el tiempo utilizado en la planificación debe ser el mínimo posible, ya que es tiempo inútil para el SO, como ocurría con el dispatcher.
- **Tiempo de respuesta aceptable:** Es el tiempo que transcurre desde que se crea o lanza un proceso (estado Nuevo) hasta que se produzca su primera interacción es estado Ejecutando.
- **Ocupación de los recursos:** Se debe maximizar el uso de los recursos del sistema para evitar su bloqueo.
- **Degradación aceptable:** Si hay muchas peticiones de nuevos procesos o mucho uso de la CPU, el algoritmo de planificación debe procurar una degradación del rendimiento de forma suave y sin que ocurra una caída global. El planificador a largo plazo limita el número de procesos en RAM y el de medio plazo se encarga del swapping.

2. CRITERIOS PARA LA PLANIFICACIÓN

Los criterios sirven para comparar el rendimiento de los diversos algoritmos de planificación:

- **Tiempo de retorno, de residencia, o de estancia (turnaround time):** Es el tiempo que transcurre desde el momento en que se crea un trabajo, estado Nuevo, hasta el momento en que se completa. Estado Saliente. Incluye el tiempo de uso de CPU, el de espera, tiempos de E/S...
- **Tiempo de espera (waiting time):** Es la suma de tiempos que el proceso está esperando a ser servido, es decir, la suma de tiempos de permanencia en estados distintos al de Ejecución. Objetivo: reducir el tiempo medio de espera.
- **Utilización de la CPU, eficacia, o tiempo de servicio (CPU utilization):** Es el porcentaje de tiempo que está ocupada la CPU de forma útil. Es la suma de tiempos de un proceso está en la CPU, el tiempo que necesita para ejecutarse completamente.

3. MODOS DE DECISIÓN

Los algoritmos de planificación se pueden dividir en dos categorías con respecto a la forma en que manejan los instantes de tiempo en que se ejecuta un proceso:

- No apropiativos o sin expulsión:** El planificador selecciona un proceso para ejecutarlo y después deja que se ejecute en CPU hasta que el propio se bloquee, termina o se produce una interrupción que no tenga que ver con la planificación. No existe rodaja de tiempo.
- Apropiativos o con expulsión:** El planificador selecciona un proceso para ejecutarlo y la decisión de expulsarlo de la CPU se toma si sucede algún tipo de evento debido a la política de planificación como, por ejemplo:
 - o El proceso se ha ejecutado por un máximo de tiempo fijo (quantum) o rodaja de tiempo). Si sigue en ejecución, se pasa a la lista de Listos y el planificador selecciona a otro proceso.
 - o Un nuevo proceso con mayor prioridad llega a la lista de Listos, desde Nuevo o Bloqueado.

Como ventaja pueden proporcionar mejor servicio a la población total de procesos, ya que previenen que cualquier proceso pueda monopolizar el procesador durante mucho tiempo y produzca inanición de procesos.

Como desventaja, las políticas expulsivas tienen mayor sobrecarga que las no expulsivas, ya que hay muchos más cambios de contexto.

A continuación, se reorganizan y se muestran ejemplos que darían lugar a la expulsión de un proceso de la CPU:

- Se produce una interrupción proveniente de una E/S anterior, de forma que se interrumpe la ejecución actual para tratarla.
- Si el proceso actual pasa a estado Bloqueado por:
 - o Por llamadas al sistema que provoquen E/S como `open()`, `read()`... y el proceso entra en estado Bloqueado.
 - o Ante una llamada bloqueante, como cuando un proceso se bloquea a esperar de que termine un hijo o un hilo mediante `waitpid()`, que pueden bloquear el proceso actual.

4. TIPOS DE PLANIFICACIÓN

El objetivo de la planificación es asignar procesos a ser ejecutados por el procesador a lo largo del tiempo, de forma que se cumplan los objetivos del sistema como tiempo de respuesta, rendimiento... Esta se divide en 3 funciones independientes:

- **La planificación a largo plazo:** Se realiza cuando se crea un nuevo proceso, de forma que hay que decidir si se añade un nuevo proceso al conjunto de los que están activos actualmente. Este planificador determinará qué programas se admiten en el sistema para su procesamiento, controlando el grado de multiprogramación. Cuanto mayor sea el número de procesos cargados en RAM, menor será el tiempo en que cada proceso se puede ejecutar, y este planificador puede limitar el grado de multiprogramación para satisfacer al conjunto de procesos cargados, de forma que cuando acabe uno, puede decidir si añadir trabajos Nuevos a la lista de Listos. Si el procesador excede el tiempo, el planificador puede traer procesos desde Nuevo a Listo.

- **La planificación a medio plazo:** Es parte de la función de intercambio o memoria virtual (swapping). Ayuda a decidir si un proceso se añade a memoria principal o a disco duro.
- **La planificación a corto plazo:** Sirve para decidir cuál de los procesos Listos para ejecutar es el siguiente a introducir en la CPU. Hay que expulsar al proceso actual.

En términos de frecuencia de ejecución:

- El planificador a largo plazo se ejecuta con poca frecuencia para admitir un nuevo proceso y cuando admitirlo.
- El planificador a medio plazo es poco frecuente ya que se usa para tomar decisiones de intercambio. Actualmente se usa cada vez menos ya que se dispone de una gran cantidad de memoria RAM para alojar procesos y su frecuencia de uso es muy pequeña.
- El planificador a corto plazo se ejecuta mucho más frecuentemente para saber qué proceso se ejecuta el siguiente.

5. TIPOS DE PROCESOS

La decisión de cuál es el siguiente trabajo por admitir para la CPU puede basarse en encontrar un compromiso entre procesos limitados por el procesador y procesos limitados por la E/S:

- Un proceso está limitado por el procesador si realiza mucho trabajo computacional y usa muy poco los dispositivos de E/S.
- Un proceso está limitado por la E/S si se pasa más tiempo utilizando los dispositivos de E/S que el procesador.

6. PRIMERO EN LLEGAR, PRIMERO EN SERVIRSE (FCFS)

Esta es la planificación más sencilla, también llamada **FIFO**. En el momento en que un proceso pasa a Listo, hay una sola cola y cuando el proceso actual termina, se coge el proceso que lleva más tiempo en Listo. En este algoritmo no existen las rodajas de tiempo, es un modelo de decisión no expulsivo, por lo que un proceso se puede ejecutar entero a no ser que una operación lo lleve a Bloqueado y pase a la lista de Bloqueados esperando el evento hasta que ocurre y pase a Listo. Partiendo del siguiente ejemplo:

Proceso	Tiempo de Llegada	Tiempo de Servicio (T_s)	Tiempo de Comienzo	Tiempo de Finalización	Tiempo de Estancia (T_r)	T_f/T_s
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1,99
Media					100	26

Podemos analizar las siguientes características en FCFS:

1. El tiempo de estancia para el proceso Y es muy grande en comparación al resto ya que es 100 veces mayor que el tiempo requerido para su procesamiento. Este inconveniente sucederá siempre que llegue un proceso corto a continuación de un proceso largo. (Y llega en el 2 y es largo y retrasa a X que es corto).
2. Otro inconveniente es que tiende a favorecer procesos limitados por el procesador sobre los limitados por la E/S:
 - o Si un proceso limitado por el procesador llega primero y está ejecutando, el resto de los procesos debe esperar y aunque haya sólo uno limitado por el procesador hace que se paren todos los demás.
 - o Un proceso limitado por E/S que pase a Bloqueado al principio de la ejecución, no volverá a entrar en Ejecución hasta que se desbloquee.
3. Los procesos largos no van mal, ya que Z tiene casi el doble de tiempo de estancia que Y, pero el normalizado está por debajo de 2. Sigue cuando hay procesos largos tras procesos cortos.

FCFS no es una alternativa atractiva por sí misma para un sistema, aunque a veces se combinan con otras políticas.

6.1 EJEMPLO

Un ejemplo del algoritmo de planificación FCFS:

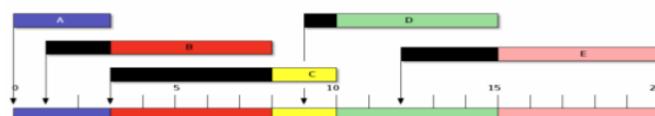
La tabla de información de los procesos es:

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	3	3-0=3	3-3=0
B	1	5	3	8	8-1=7	7-5=2
C	3	2	8	10	10-3=7	7-2=5
D	9	5	10	15	15-9=6	6-5=1
E	12	5	15	20	20-12=8	8-5=3

El esquema de ejecución de los procesos en el sistema es:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	x	x	x																	
B				x	x	x	x	x												
C									x	x										
D										x	x	x	x	x						
E															x	x	x	x	x	

Partiendo de:



7. TURNO ROTATORIO (ROUND ROBIN)

Para reducir el castigo de los trabajos cortos después de largos se usa la expulsión basándose en el reloj. La más sencilla es el turno rotatorio o planificación cíclica.

Se genera una interrupción de reloj cada cierto intervalo de tiempo, que hace que el proceso que está en ejecución se sitúe en Listos y se coge el siguiente según el FCFS. A cada proceso se le da una rodaja de tiempo antes de ser expulsado. Si antes de finalizar invoca a una operación de E/S, el proceso no cumpliría la rodaja entera.

El tema clave de diseño es la longitud del quantum de tiempo a ser utilizado:

- Si es muy pequeño, el proceso se moverá por el sistema rápido. Existe una sobrecarga de procesamiento debido al manejo de la interrupción de reloj y por el dispatcher. Por esto, se deben evitar los quantums de tiempo pequeños.
- Si es mayor que el proceso más largo, se puede provocar inanición ya que el último proceso tardaría mucho en entrar en CPU favoreciendo a los largos y perjudicando a los cortos.

Otra desventaja es que trata de forma desigual a los procesos limitados por el procesador y a los limitados por la E/S. Los limitados por el procesador van a utilizar su rodaja de tiempo entera mientras que los de E/S tiene ráfagas de procesador más cortas, ya que antes de que acaben su quantum invocan una operación E/S que los lleva a Bloqueado. Hay que tener en cuenta la siguiente suposición: si en un mismo instante de tiempo un proceso sale de la CPU y otro llega a Listos, el que entra se pone delante del que sale.

7.1 EJEMPLO 1

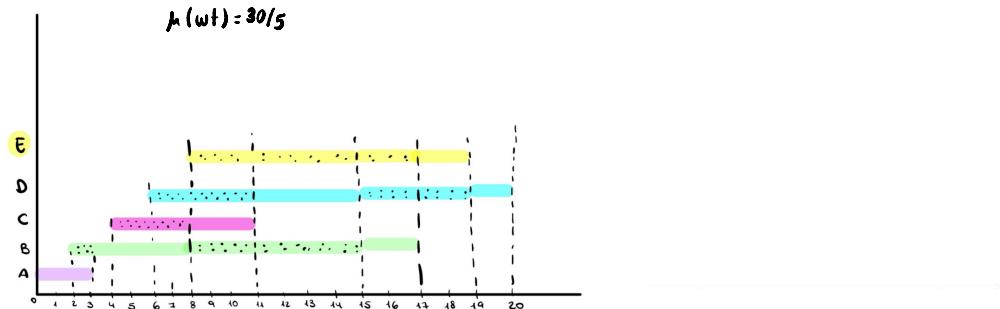
Para q=4 sería:

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	3	3	0
B	2	6	3	17	15	9
C	4	4	7	11	7	3
D	6	5	10	20	14	9
E	8	2	17	19	11	9

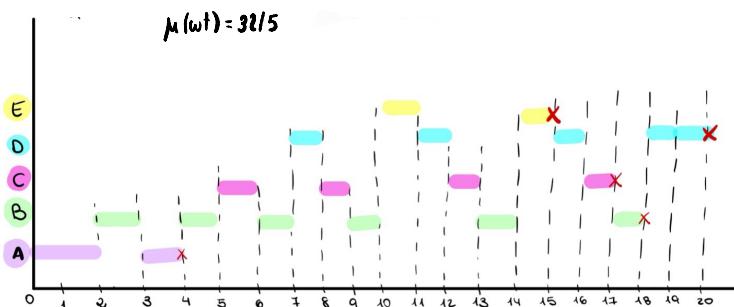
Para q=1 sería:

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	4	4	1
B	2	6	2	18	16	10
C	4	4	5	17	13	9
D	6	5	7	20	14	9
E	8	2	10	15	7	5

El esquema de $q=4$ es:



El esquema de $q=1$ es:



7.4 TURNO ROTATORIO VIRTUAL (VIRTUAL ROUND ROBIN)

NO
ENTRA

Existe un refinamiento de la planificación en turno rotatorio llamado turno rotatorio virtual que evita que los procesos limitados por E/S no se vean perjudicados.

Los nuevos procesos que llegan se unen a la cola de Listos gestionada con FCFS. Cuando expira el tiempo de ejecución, vuelve a la cola de Listos. Cuando se bloquea un proceso por E/S, se une a la cola de Bloqueados.

Se crea una cola auxiliar FCFS a la que se mueven los Bloqueados en una E/S. Cuando se va a tomar un proceso por el planificador a corto plazo, los procesos de la auxiliar tienen preferencia sobre los Listos y cuando se activa un proceso desde la auxiliar, se ejecuta por un tiempo no superior a lo que resta de su rodaja de tiempo:

- Si lo que le resta es 0, es porque ha cumplido antes su rodaja entera, por lo que volverá a tener una nueva rodaja. Se usa esta sino se indica lo contrario.
- Otro esquema es que si el proceso que sale de Bloqueados le resta 0, pase a Listos en lugar de a la auxiliar.

7.4.1 Ejemplo 1

Teniendo quantum de tiempo $q=3$ y teniendo en cuenta que en el ciclo 4, el proceso B realiza una operación de E/S que dura hasta el 5 y vuelve a estar disponible para ejecutarse en el ciclo 9. Además, en el ciclo 15, el proceso D realiza una operación de E/S que dura hasta el 16 y vuelve a estar disponible para su ejecución en el 17.

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	3	3-0=3	3-3=0
B	2	6	3	19	19-2=17	17-6=11
C	4	4	5	15	15-4=11	11-4=7
D	6	5	8	20	20-6=14	14-5=9
E	8	2	12	14	14-8=6	6-2=4

El esquema de los procesos en el sistema es:

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	x	x	x																	
B				x	x						x					x	x	x		
C					x	x	x							x						
D							x	x	x					x				x		
E											x	x								

8. PLANIFICACIÓN MEDIANTE COLAS DE PRIORIDADES MULTINIVEL

En muchos sistemas, a cada proceso se le asigna una prioridad y una cola multinivel para que el planificador elija un proceso de prioridad mayor sobre uno de menor. Esta es establecida por el sistema en función de parámetros como el propietario del proceso...

Cuando se va a realizar una selección en la planificación, el planificador comienza por la cola de Listos con la prioridad más alta (CL0), si hay más de uno, se selecciona uno mediante FIFO. Si CL0 está vacía pasa a CL1 y así sucesivamente.

Esta política es expulsiva por rodaja de tiempo o quantum, de forma que un proceso se expulsa de la CPU si cumple su rodaja de tiempo, se produce una interrupción o se invoca una operación que lo lleve a Bloqueado.

Si la expulsión de un proceso se realizase antes de cumplir su rodaja de tiempo, el proceso no cumplirá la rodaja entera y si volviese a Listos no ejecutaría el tiempo restante, sino que comenzaría la rodaja de nuevo.

Habrá tantas colas en el sistema como prioridades, a no ser que se agrupen rangos de prioridades en colas. Como la prioridad no cambia durante la ejecución del proceso se llama política de prioridades estáticas.

Otro problema de las colas multinivel es que un proceso con baja prioridad puede llegar a inanición si hay continuamente procesos con mayor prioridad listos para ejecutarse. Hay que tener en cuenta lo siguiente:

- En el caso de que el proceso salga de la CPU a la vez que llega un nuevo proceso, el nuevo proceso se añade a Listos antes de que termine.
- Si un proceso termina y no queda ningún proceso, este sigue en CPU.

8.1 EJEMPLO 1

Si se considerase q=2, la tabla es la siguiente suponiendo que hay 3 colas, una por prioridad y teniendo en cuenta que el valor 1 da más prioridad que el 2:

Proceso	Prioridad	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	2	0	3	0	7	7-0=7	7-3=4
B	2	1	5	2	10	10-1=9	9-5=4
C	1	3	2	4	6	6-3=3	3-2=1
D	3	9	5	10	20	20-9=11	11-5=6
E	1	12	5	12	17	17-12=5	5-5=0

El esquema de ejecución es:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	x	x	w	w	w	w	x													
B		w	x	x	w	w	w	x	x	x										
C			w	x	x															
D									w	x	x	w	w	w	w	w	x	x	x	
E												x	x	x	x	x				

$$M(wt) = 15 / 5 = 3$$

9. COLAS DE PRIORIDAD MULTINIVEL RETROALIMENTADA (FEEDBACK)

En esta política de planificación hay fijado un número máximo de colas con prioridad y funciona exactamente igual que en el punto anterior, pero con la siguiente modificación:

Si un proceso se expulsa de una cola, aunque sea por una E/S, automáticamente pasará a la siguiente cola con menos prioridad. Si no ha cumplido su rodaja de tiempo no ejecutará lo que reste, sino que comienza una rodaja nueva. Cuando un proceso cambia su nivel de prioridad se llama prioridades dinámicas. La cola de menor prioridad se trata con una política de turno rotatorio ya que no puede descender más.

Los problemas que presenta esta política son que un esquema de colas multinivel con valores de q muy pequeños puede provocar que:

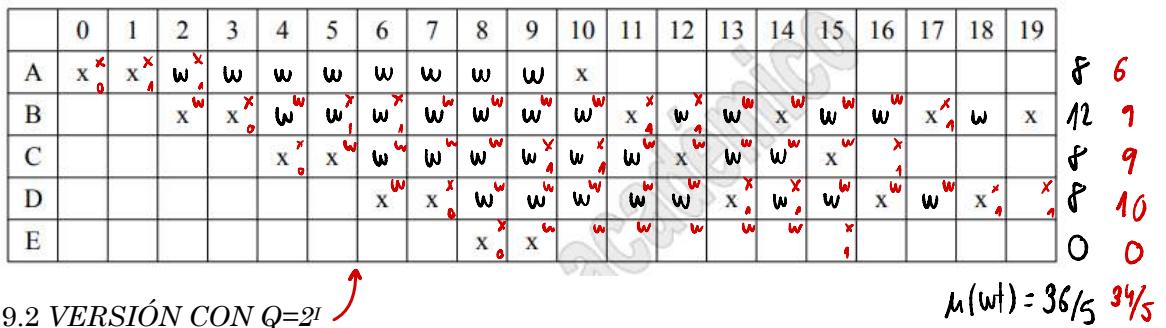
- El tiempo de estancia para procesos más largos se puede alargar de forma alarmante.
- Puede ocurrir inanición para procesos más largos si están entrando nuevos trabajos frecuentemente en el sistema ya que el planificador siempre empieza por CL0.

9.1 EJEMPLO 1

Este ejemplo muestra la resolución de un problema con q=1 y 3 colas, la última de ellas con feedback:

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	11	11-0=11	11-3=8
B	2	6	2	20	20-2=18	18-6=12
C	4	4	4	16	16-4=12	12-4=8
D	6	5	6	19	19-6=13	13-5=8
E	8	2	8	10	10-8=2	2-2=0

El esquema de ejecución de los procesos en el sistema es:



Para compensar el problema del quantum $q=1$, se pondrían variar los tiempos de expulsión de cada cola, de manera que un proceso de la cola CL0 tiene una rodaja de una unidad de tiempo; el de la CL1 dos unidades de tiempo... a un proceso de la cola CL*i* se le permitiría ejecutar 2^i ($q=2^i$) unidades de tiempo antes de ser expulsado.

10. PRIMERO EL PROCESO MÁS CORTO (SHORTEST PROCESS NEXT-SPN)

También es conocido como Shortest job next (SJN) y Shortest job first (SJF).

SPN es una política no expulsiva en la que se selecciona el proceso con el tiempo de procesamiento más corto esperado, es decir, el que necesite menos CPU. De esta manera un proceso corto se situará en la cabeza de la cola de Listos. Una vez dentro de la CPU se ejecuta entero sino se bloquea, aunque lleguen nuevos procesos, los tiempos no se recalculan, esto se hace cuando el actual termine para ver cuál es el siguiente más corto.

En el caso de que un proceso se bloquee, pasaría a la lista de Bloqueados en espera de un evento, igual que en FCFS. Cuando ocurra pasará a Listos con un tiempo de servicio igual al restante que le quede por cumplir.

Si 2 o más procesos tienen el mismo tiempo de servicio se le da prioridad al que lleva más tiempo en el sistema (FIFO). Es necesario contar con información por anticipado acerca del tiempo de CPU que requieren los procesos que están en Listos. En las políticas anteriores no se tenía en cuenta el Tº de CPU para tomar una decisión por parte del planificador. Esta información se suele estimar mediante:

- Modelos matemáticos que asignan un tiempo de servicio a un proceso dependiendo de su tipo...

Al trabajar con estimaciones, el éxito de la planificación dependerá de lo adecuado que sea el modelo matemático. Con SPN se pueden dar estos problemas:

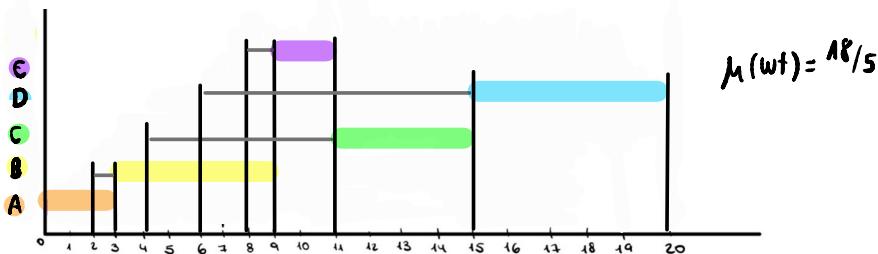
- Hay posibilidad de inanición para los procesos más largos si hay llegadas de procesos más cortos mientras el largo está encolado.
- Hay posibilidad de penalizar fuertemente a un proceso corto tras uno largo que ya está ejecutando debido a la carenza de expulsión.

10.1 EJEMPLO 1

La tabla muestra el resultado con procesos limitados por el procesador:

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	3	3-0=3	3-3=0
B	2	6	3	9	9-2=7	7-6=1
C	4	4	11	15	15-4=11	11-4=7
D	6	5	15	20	20-6=14	14-5=9
E	8	2	9	11	11-8=3	3-2=1

El esquema de ejecución de los procesos en el sistema es:



11. MENOR TIEMPO RESTANTE (SHORTEST REMAINING TIME-SRT)

Si la SPN se implementa con una decisión expropiativa, se podría expulsar procesos sin tener en cuenta su rodaja de tiempo. Si un proceso se une a la lista de procesos Listos con menor CPU restante, se le quita la CPU al proceso actual para asignársela al nuevo que es más corto, el planificador debe tener una estimación del tiempo de servicio o CPU de los procesos que pasan a la lista de Listos por primera vez.

La diferencia con respecto al SPN es que en SRT la lista de Listos se comprueba cada vez que llegan procesos a ésta y no una vez se ha ejecutado el proceso actual.

Si 2 o más procesos tienen el mismo tiempo de servicio restante se le da prioridad a 1 que lleva más tiempo en Listos (FIFO). Existe riesgo de inanición para los procesos largos.

No se generan interrupciones adicionales por finalización de rodaja de tiempo, reduciéndose la sobrecarga sino llegan procesos con menor tiempo de servicio.

Estos tiempos deben ser almacenados conforme transcurren, generando sobrecarga y habría que interrumpir el proceso actual en ejecución para calcular el tiempo restante de los nuevos procesos que llegan a Listos.

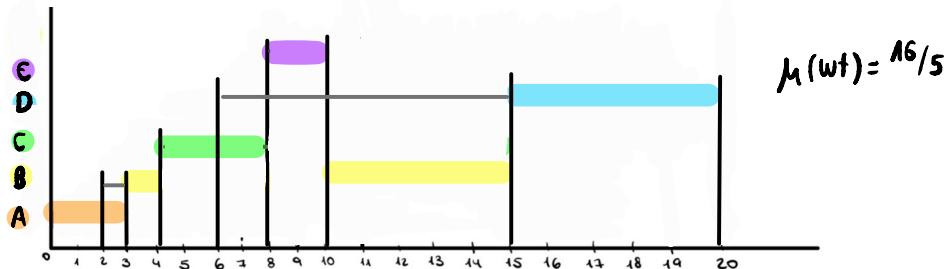
SPN y SRT no se pueden implementar en sistemas en los que no se sepa a priori el tiempo de CPU que requieren los procesos situados en Listos.

11.1 EJEMPLO 1

La tabla muestra el resultado con procesos limitados por el procesador:

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	3	3-0=3	3-3=0
B	2	6	3	15	15-2=13	13-6=7
C	4	4	4	8	8-4=4	4-4=0
D	6	5	15	20	20-6=14	14-5=9
E	8	2	8	10	10-8=2	2-2=0

El esquema de ejecución de los procesos en el sistema es:



En el instante 10, tanto el proceso D como el B tienen el mismo tiempo restante pero como B lleva más tiempo, se le da a este la prioridad.

12. COMPARACIÓN DE RENDIMIENTO

El rendimiento de las políticas de planificación es un factor crítico en su elección, pero es imposible hacer una comparación definitiva ya que el rendimiento depende de multitud de factores como la eficiencia de la planificación, la distribución de los tiempos de servicio de varios procesos... Por tanto, dependiendo de los factores, lo ideal sería que el sistema aplicase una política de planificación u otra.

No es lo mismo un sistema de propósito general que sistemas de cómputo de propósito específico, donde sabemos qué tipo de procesos va a haber en el sistema y para qué se va a utilizar. En éste último caso la elección de la planificación es más sencilla y concreta.