TEMA 1. ESTRUCTURA DE LOS SO

Sistema monolítico

Los **sistemas monolíticos** son aquellos en los que su centro es un grupo de estructuras fijas, las cuales funcionan entre sí, para poder tener esta estructura, las diferentes partes del kernel son compiladas por capas.

Los sistemas monolíticos se describen en **3 procesos principales**:

- Un Programa principal que invoca el procedimiento de servicio solicitado.
- Un Conjunto de procedimientos de servicio que llevan a cabo las llamadas del sistema.
- Un Conjunto de procedimientos de utilería que ayudan a los procedimientos del servicio.

Los núcleos monolíticos proporcionan la mayor parte de las funcionalidades propias del sistema operativo, incluyendo la planificación, los sistemas de ficheros, las redes, los controladores de dispositivos, la gestión de memoria y otras funciones. El SO está formado por un conjunto de procedimientos de forma que cada uno puede llamar a los demás cuando lo necesite. Estos procedimientos se ejecutan en modo supervisor. Normalmente, un núcleo monolítico se implementa como un único proceso con todos los elementos compartiendo el mismo espacio de direcciones.

El **problema** de los núcleos monolíticos es que son difíciles de comprender, modificar y mantener. No son fiables: un error en alguna parte puede provocar la caída del sistema.

Se podría implementar como un **sistema por capas**, donde cada capa es una máquina más abstracta para la capa superior. Por modularidad, las capas se seleccionan para que cada una utilice funciones sólo de las capas inferiores.

Ejemplo de sistema por capas: **El Sistema THE**

El sistema estaba compuesto de una serie de procesos secuenciales.

Nivel 5	Programas de usuario
Nivel 4	Búfering para dispositivos de E/S
Nivel 3	Manejador de consola del operador
Nivel 2	Gestión de memoria
Nivel 1	Planificación de la CPU
Nivel 0	Hardware

Problemas de THE:

- Los sistemas de capas deben ser jerárquicos pero los sistemas reales son

más complejos. Por ejemplo: el sistema de archivos podría ser un proceso en la capa de memoria virtual y la capa de memoria virtual podría usar archivos como almacén de apoyo de E/S.
- Sobrecarga de comunicaciones entre procesos de distintas capas
- A menudo, los sistemas están modelados con esta estructura pero no están así construidos

Microkernel

Una **arquitectura micronúcleo** asigna sólo unas pocas funciones esenciales al núcleo, incluyendo los espacios de almacenamiento, comunicación entre procesos (IPC), y la planificación básica. Ciertos procesos proporcionan otros servicios del sistema operativo, algunas veces denominados servidores, que ejecutan en modo usuario y son tratados como cualquier otra aplicación por el micronúcleo

El sistema operativo se reduce a un núcleo mínimo, se implementan la mayoría de las funciones del SO como procesos de usuario, lo que proporciona mayor flexibilidad. Para solicitar un servicio, el proceso de usuario (cliente) envía un mensaje al proceso servidor, que realiza el servicio y devuelve al cliente una respuesta, provocando una mayor sobrecarga por en envío/recepción de mensajes.

Beneficios:

- Su arquitectura facilita la extensibilidad, permitiendo agregar nuevos servicios en la misma área funcional.
- No solo se pueden añadir nuevas características al SO, además las características existentes se pueden eliminar para realizar una implementación más pequeña y eficiente.
- Todo o gran parte del código específico del procesador está en el microkernel.

Una de las **desventajas** más conocida del microkernel es la del rendimiento. Esto depende del tamaño y la funcionalidad del microkernel.

Multithreading

Multithreading es una técnica en la cual un proceso, ejecutando una aplicación, se divide en una serie de hilos o threads que pueden ejecutar concurrentemente.

- **Thread o hilo**. Se trata de una unidad de trabajo. Incluye el contexto del procesador (que contiene el contador del programa y el puntero de pila) y su propia área de datos para una pila (para posibilitar el salto a subrutinas). Un hilo se ejecuta secuencialmente y se puede interrumpir de forma que el procesador pueda dar paso a otro hilo.
- **Proceso**. Es una colección de uno o más hilos y sus recursos de sistema asociados (memoria, conteniendo tanto código, como datos, ficheros abiertos

y dispositivos). Esto corresponde al concepto de programa en ejecución. La técnica multithreading es útil para las aplicaciones que llevan a cabo un número de tareas independientes.

Máquinas virtuales

Una **máquina virtual** es un software que implementa una máquina virtual (igual o distina a la máquina real). Se trata de abstraer el hardware de la computadora formando entornos de ejecución diferentes, creando la ilusión de que cada entorno de ejecución está en una computadora diferente. La máquina virtual no proporciona funcionalidades adicionales sino que proporciona una interfaz "idéntica" al hardware básico. Una petición de servicio es atendida por la copia de SO sobre la que se ejecuta. La capacidad de procesamiento actual mitiga la ineficiencia de las Máquinas Virtuales. Los procesadores más actuales incluyen soporte para la misma.

Beneficios:

- Más seguras, no es posible la compartición directa de recursos.
- Investigación y desarrollo de sistemas operativos.

Sistemas Operativos de propósito específico

Sistemas Operativos de tiempo real

La computación de tiempo real puede definirse como aquella en la que la corrección del sistema depende no sólo del resultado lógico de la computación sino también del momento en el que se producen los resultados.

En un **sistema de tiempo real** se producen unas tareas que dan soluciones a problemas que vienen del mundo exterior y que se producen en un espacio de tiempo determinado, pudiéndose establecer así un margen de tiempo límite para realizar las tareas. De este modo se pueden dividir las tareas en tareas de **tiempo real duro** si tienen que realizarse en un tiempo determinado y un exceso de tiempo en dicha actividad puede suponer un error grave para el sistema, o tareas de **tiempo real blando** cuyo plazo de tiempo no es estricto y la tarea se puede seguir ejecutando después de haber cumplido el plazo sin desencadenar errores graves.

Estas tareas se distinguen a su vez dada la periodicidad de su ejecución, de este modo hay **tareas aperiódicas** si su ejecución es única y está restringida a un plazo de tiempo, o **tareas periódicas** si se repiten n veces en un espacio de tiempo o una vez cada período t.

Los sistemas operativos de tiempo real pueden ser caracterizados por tener requisitos únicos en ***cinco áreas generales:***

- Determinismo
- Reactividad
- Control del usuario
- Fiabilidad
- Operación de fallo suave

El **determinismo** (tiempo de SO en reconocer la interrupción) nos dice que unas tareas concretas van a tardar un tiempo concreto en ser finalizadas, un sistema no puede ser completamente determinista, pues se ve limitado por sus recursos y por la prioridad de las tareas a ejecutar, se centra en el tiempo previo al reconocimiento previo de una instrucción.

La **reactividad**, relacionada con el determinismo, se enfoca en el tiempo después del reconocimiento que tarda el SO en servir a esa interrupción, además incluye algunos aspectos como:

1. La cantidad de tiempo necesario para ejecutar la rutina de servicio de la interrupción actual, RSI (si es necesario cambiar de contexto requerirá un tiempo adicional).
2. El tiempo que requiere para procesarla, que se define a nivel hardware.
3. El efecto anidamiento. Si una interrupción puede ser interrumpida por otra.

El **control del usuario** es generalmente mucho mayor en un sistema operativo de tiempo real que en sistemas operativos ordinarios. El usuario tiene un control sobre la prioridad de la tarea. El usuario distingue las tareas y se encarga de gestionar algunas características de gestión de memoria.

La **fiabilidad** en sistemas de tiempo real es mucho más trascendental que en los sistemas que no son de tiempo real, pues en estos se puede solucionar el problema re-arrancando el sistema, en los de tiempo real la degradación de los servicios ofrecidos a un sistema que responde a tareas en tiempo real puede ser mucho más catastrófica.

La **operación de fallo suave** es una característica de los sistemas para responder ante un fallo y ser capaz de minimizar las pérdidas sufridas, manteniendo la consistencia de ficheros y datos cuando sea posible. Además un sistema se considera estable cuando a pesar de no ser capaz de responder a todas las peticiones, es capaz de responder a las tareas más críticas.

Para cumplir los requisitos precedentes, los sistemas operativos de tiempo real incluyen de forma representativa las siguientes características:

- Cambio de proceso o hilo rápido.
- Pequeño tamaño (que está asociado con funcionalidades mínimas).
- Respuesta rápida a interrupciones externas.

- Multitarea con herramientas para la comunicación entre procesos como semáforos, señales y eventos.
- Utilización de ficheros secuenciales especiales que pueden acumular datos a alta velocidad.
- Planificación expulsiva basada en prioridades.
- Minimización de los intervalos durante los cuales se deshabilitan las interrupciones.
- Primitivas para retardar tareas durante una cantidad dada de tiempo y para parar/retomar tareas.
- Alarmas y temporizaciones especiales.

En la mayoría de los casos los sistemas no son capaces de lidiar con todas las tareas asignadas, por lo que una de las principales áreas de trabajo hoy en día son los planificadores de tiempo real. Lo importante es que las tareas de tiempo real duro se completen y finalizar el máximo número de tareas de tiempo real blando.

Estructuras distribuidas

Hay una tendencia al proceso de datos distribuidos. Los procesadores, datos y procesamiento de datos pueden estar diseminados por toda la organización. Implica dividir funciones y organizar las bases de datos control de dispositivos,... Los computadores dependen de su asociación con los servidores. Existen distintas estructuras distribuidas: arquitectura multicomputador, sistemas operativos de red y sistemas operativos distribuidos.

Arquitectura Multicomputador

La **arquitectura de comunicaciones** es un software que da soporte a un grupo de computadores independientes, en red. Proporciona soporte para aplicaciones distribuidas, tales como correo electrónico, transferencia de ficheros y acceso a terminales remotos. Cada cual puede tener su propio SO y sólo se pueden comunicar directamente por deseo expreso.

Sistemas Operativos de Red

Un **sistema operativo de red** normalmente lo componen un único usuario con una o varias máquinas de servidores, que proporcionan acceso a servicios y aplicaciones, el usuario conoce la existencia de múltiples computadores y debe trabajar con ellos de forma explícita. Habitualmente se utiliza una arquitectura de comunicaciones común para dar soporte a estas aplicaciones de red. Lo que les diferencia de los SO de un solo procesador es la necesidad de software especial como: controlador de interfaz de la red; programas de conexión y acceso a archivos remotos.

Sistemas operativos Distribuidos

Un **sistema operativo distribuido** es un sistema operativo común compartido por una red de computadores. A los usuarios les proporciona acceso transparente a los recursos de diversas máquinas. Un sistema operativo distribuido puede depender de una arquitectura de comunicaciones para las funciones básicas de comunicación, pero normalmente se incorporan un conjunto de funciones de comunicación más sencillas para proporcionar mayor eficiencia.

Para hacer un intercambio de información entre dos computadores, ya sea algo tan sencillo como un fichero, es necesario establecer un enlace, tanto directo como en red de comunicaciones, pero además se necesitan tareas como:

1. Que el emisor active el enlace directo o informar a la propia red de comunicaciones
2. Debe comprobar que el receptor puede recibir estos datos
3. Que existe un programa que sea capaz de recepcionar el fichero
4. Que se encargue de realizar una traducción si las representaciones de datos son incompatibles entre ambos sistemas

En relación a la comunicación de computadores y redes de computadores, hay dos conceptos de suma importancia:

- Protocolos.
- Arquitectura de comunicaciones o arquitectura de protocolos.

Un **protocolo** se utiliza para comunicar entidades de diferentes sistemas. Lo que se comunica, cómo se comunica y cuándo se comunica, debe hacerse de acuerdo a unas convenciones entre las entidades involucradas. Se pueden definir como un conjunto de reglas que gobiernan el intercambio de datos entre dos entidades. Los elementos principales de un protocolo son los siguientes:

- **Sintaxis**: Incluye cosas tales como formatos de datos y niveles de señales.
- **Semántica**: Incluye información de control para realizar coordinación y gestión de errores.
- **Temporización**: Incluye ajuste de velocidades y secuenciamiento.

Arquitectura de protocolos:

Dada la complejidad de los sistemas, en la red hay que establecer una arquitectura de protocolos, que será la encargada de realizar las comunicaciones entre máquinas dividiendo las tareas en subtareas, en módulos, que contienen claves, mandatos, registros,... otros que ese encarguen de comprobar que las comunicaciones están bien establecidas,... En general, una estructura que sea capaz de responder de manera eficiente ante las diferencias de los sistemas que se están comunicando.

Arquitectura Multiprocesador: Sistemas Operativos Paralelos

Históricamente el computador se ha considerado como una unidad de procesamiento secuencial, donde toda instrucción se sucedía una tras otra, pero este concepto ha ido cambiando, pues hoy en día, por motivos de optimización, se utilizan nuevas técnicas de paralelismo (procesamiento de funciones en varios dispositivos a la vez). En particular el *Multiprocesamiento Simétrico (SMP) y los clusters.*

ARQUITECTURA SMP

Es útil ver donde encaja la arquitectura SMP dentro de las categorías de procesamiento paralelo. La forma más común de categorizar estos sistemas es la clasificación de sistemas de procesamiento paralelo introducida por Flynn. El cual propone las siguientes categorías de sistemas de computadores:

- **Única instrucción, único flujo de datos**: *(Single instruction single data (SISD) stream).* Un solo procesador ejecuta una única instrucción que opera sobre datos almacenados en una sola memoria.
- **Única instrucción, múltiples flujos de datos** : *(Single instruction multiple data (SIMD) stream).* Una única instrucción de máquina controla la ejecución simultánea de un número de elementos de proceso. Cada elemento de proceso tiene una memoria de datos asociada, de forma que cada instrucción se ejecuta en un conjunto de datos diferente a través de los diferentes procesadores. Los procesadores vectoriales y matriciales entran dentro de esta categoría.
- **Múltiples instrucciones, único flujo de datos** : *(Multiple instruction single data (MISD) stream).* Se transmite una secuencia de datos a un conjunto de procesadores, cada uno de los cuales ejecuta una secuencia de instrucciones diferente. Esta estructura nunca se ha implementado.
- **Múltiples instrucciones, múltiples flujos de datos**: *(Multiple instruction multiple data (MIMD) stream).* Un conjunto de procesadores ejecuta simultáneamente diferentes secuencias de instrucciones en diferentes conjuntos de datos.

Un sistema con esta última organización MIMD recibe el nombre de ***cluster***, donde si cada procesador accede a una misma memoria se le conoce como **multiprocesador de memoria compartida**. Y en estos se usa una estructura de maestro/esclavo para controlar y organizar, pero esto tiene algunas desventajas, por ejemplo un fallo en el maestro puede derrumbar todo el sistema, o las limitaciones en este pueden afectar a los que se encuentran subyugados.

ORGANIZACIÓN SMP

La organización de procesamiento multisimétrico permite que se planifiquen todas las tareas y se dividan en hilos previos a su procesamiento, cada proce-

sador contiene sus propias unidades, pero se encuentran unidas a una memoria compartida

DISEÑOS DE SISTEMAS OPERATIVOS MULTIPROCESADOR

El diseño de un SO permite encargarse de todas las peticiones sin que el usuario tenga que responder a estas, por lo tanto este diseño debe de tener unas características especiales como:

- **Procesos o hilos simultáneos concurrentes**. Las rutinas del núcleo necesitan ser reentrantes para permitir que varios procesadores ejecuten el mismo código del núcleo simultáneamente. Debido a que múltiples procesadores pueden ejecutar la misma o diferentes partes del código del núcleo, las tablas y la gestión de las estructuras del núcleo deben ser gestionas apropiadamente para impedir interbloqueos u operaciones inválidas.
- **Planificación**. La planificación se puede realizar por cualquier procesador, por lo que se deben evitar los conflictos. Si se utiliza multihilo a nivel de núcleo, existe la posibilidad de planificar múltiples hilos del mismo proceso simultáneamente en múltiples procesadores.
- **Sincronización**. Con múltiples procesos activos, que pueden acceder a espacios de direcciones compartidas o recursos compartidos de E/S, se debe tener cuidado en proporcionar una sincronización eficaz. La sincronización es un servicio que fuerza la exclusión mutua y el orden de los eventos.
- **Gestión de memoria**. La gestión de memoria en un multiprocesador debe tratar con todos los aspectos encontrados en las máquinas uniprocesador. Además, el sistema operativo necesita explotar el paralelismo hardware existente, como las memorias multipuerto, para lograr el mejor rendimiento. Los mecanismos de paginación de los diferentes procesadores deben estar coordinados para asegurar la consistencia cuando varios procesadores comparten una página o segmento y para decidir sobre el reemplazo de una página.
- **Fiabilidad y tolerancia a fallos**. El sistema operativo no se debe degradar en caso de fallo de un procesador. El planificador y otras partes del sistema operativo deben darse cuenta de la pérdida de un procesador y reestructurar las tablas de gestión apropiadamente.

Resúmenes Tema 2

DGIIM

Introducción

Ejecución del S.O.

El sistema operativo se puede ejecutar de dos formas:

- Núcleo sin procesos. Es el modelo tradicional, las llamadas al sistema requieren guardar el contexto y pasar el control al kernel. En este caso, los procesos son solo los programas de usuario y el código del sistema operativo se ejecuta de forma independiente en modo *privilegiado*
- Como un proceso de usuario. El sistema operativo se ejecuta de forma virtual como si fuera un proceso. Si ocurre una interrupción, el procesador pasa a modo kernel(cambio de modo) pero no es necesario hacer un cambio de contexto, basta con guardar la información

Cambio de contexto

El cambio de contexto es la acción de cambiar la CPU de estar ejecutando un proceso a otro. Para esto, previamente se debe haber salvado la información del primer proceso(PC,SP, registros...) en su PCB para no perder esa información.

Procesos

Un proceso es un programa en ejecución.

1. Creación de un procesos. Al crear un proceso, se le asigna el espacio de direcciones que va a usar y se crean las estructuras de datos para administrarlo. Para crearlo verdaderamente, se realiza el siguiente proceso:

a) Se asigna un PID único.

b) Se reserva espacio para el proceso. (Para el espacio de direcciones privado y compartido, para la pila de usuario, PCB)

c) Se inicializa el PCB. Suele ser a 0 menos el PC y los punteros de pila del sistema. El proceso no debe poseer ningún recurso a menos que haya una indicación explícita o sea heredado del padre.

d) Se inserta en la tabla de procesos y se introduce en una cola de planificación.

e) Se crean otras estructuras de datos necesarias y se determina su prioridad.

Algunos casos comunes en los que se crea un proceso son: en respuesta a admisión y recepción de un trabajo(sistema batch), al conectar un usuario se crea un proceso que ejecuta el intérprete de órdenes(sistemas interactivos), para realizar un servicio solicitado por un proceso de usuario, o cuando un padre crea un proceso hijo.

Cuando un proceso crea a otro se deben tratar varios puntos. En cuanto a **recursos**, puede ocurrir que compartan todos los recursos, ninguno o un subconjunto. En cuanto a **ejecución**, ocurre que o bien se ejecutan a la vez o bien el padre espera a que el hijo termine. En cuanto al **espacio de direcciones** puede ser que el hijo sea un duplicado del padre o que el hijo tiene un programa que lo carga.

2. Terminación de procesos.

Puede ocurrir cuando:

a) El proceso solicita al SO su finalización. Entonces envía sus datos al padre y libera los recursos.

b) El padre aborta la ejecución de sus hijos si el hijo sobrepasa los recursos, su tarea ya no es necesaria o el padre va a finalizar

c) Lo aborta el SO pues se ha dado un fallo.

Hebras

Una **hebra** es una unidad básica de uso de la CPU. Contiene su PC, registros, espacio de pila y estado. Además, comparte con sus hebras pares una tarea que tiene su código, datos y recursos del SO.

Si un proceso es monohebra, le basta con su PCB, su espacio de direcciones y las pilas necesarias. Si es multihebra, necesita un PCB para cada hebra y las pilas para cada hebra.

Las hebras tienen como **ventajas** un mayor rendimiento y mejor servicio pues el tiempo de cambio de contexto, creación y terminación es menor, las hebras pueden ejecutarse de forma independiente y la comunicación entre hebras de la misma tarea se hace en memoria compartida.

Existen varios tipos de hebras:

- Hebras de usuario: La gestión la hace la aplicación, el núcleo no conoce que existen estas hebras. Son implementadas mediante una biblioteca en el nivel de usuario.
- Hebras kernel: Son gestionadas por el núcleo(que mantiene la información del contexto de todo el proceso y de cada hebra) y el SO proporciona llamadas a sistemas para trabajar con ellas. La unidad de planificación es la hebra y las funciones del núcleo pueden ser multihebras.

Las **ventajas** de las de tipo usuario frente a las de tipo núcleo son que no hay muchos cambios de modo, que existe una planificación para hebras distinta a la del SO y que se pueden ejecutar en cualquier SO sin hacer cambio en el nucleo.

Como **desventajas** tenemos que al bloquearse una hebra se bloquean todas las del proceso y que no se aprovechan las ventajas de un sistema multiprocesador, pues un proceso está asociado a un único procesador.

- Hebras híbridas(Hebras a nivel de kernel y de usuario): La creación y mayor parte de planificación y sincronización se hace en espacio de usuario. Las hebras de una aplicación se

relacionan con varias del núcleo y se pueden paralelizar para que si una hebra se bloquea no se bloquee todo el proceso.

Planificación

El SO tiene una colección de colas con el estado de todos los procesos. Suele haber una cola por estado. Al cambiar el estado de un proceso, su PCB se retira de una cola y se mete en otra.

El diagrama de estados por lotes es:

Diagrama de Estados

Colas de Estados

- Cola de trabajos: Pendientes de ser admitidos
- Cola de preparados: Residen en memoria principal esperando a ejecutarse.
- Cola(s) de bloqueados: Que esperan un suceso o una E/S

Planificador.

El planificador controla el uso de un recurso. Se encarga de asignar qué procesos van a ser ejecutados para cumplir los objetivos del sistema. Puede ser:

- **a largo plazo**, que selecciona los procesos que van a la cola de preparados (a memoria o a disco). Permite controlar el grado de multiprogramación y se invoca poco frecuentemente.
- **a corto plazo** que selecciona el siguiente proceso que se ejecuta y le asigna CPU. También bloquea los procesos en ejecución. Es el más frecuente. Trabaja con la cola de preparados.
- **a medio plazo** realiza una función de intercambio entre los de memoria y los de disco. Esto puede mejorar la mezcla de procesos o ayuda a cambiar los requisitos de memoria(realiza un intercambio o swapping).

Clasificación de procesos

- Limitados por E/S (o cortos): Dedican más tiempo a E/S que a cómputo, muchas ráfagas de CPU cortas y largos períodos de espera.
- Limitados por CPU(largos): Más tiempo en computación que en E/S. Al revés que los cortos.

El planificador a largo plazo debe hacer una buena mezcla de trabajos pues si no puede ocurrir que si todos los trabajos están limitados por E/S la cola de preparados estará casi siempre vacía y si todos los procesos están limitados por CPU la cola de E/S estará casi siempre vacía.

Dispatcher

El despachador es un módulo del SO que da el control de la CPU al proceso seleccionado por el planificador a corto plazo.Involucra cambio de contexto, cambio de modo a usuario y salto a la posición de memoria adecuada para la reanudación.

La latencia de despacho es el tiempo que tarda un despachador en cambiar de proceso ejecutándose.

El Despachador se activa cuando:

1. Un proceso no quiere seguir ejecutándose(acaba o se bloquea).
2. Un elemento del SO determina que el proceso no puede seguir ejecutándose(E/S)
3. El proceso agota el quantum de tiempo asignado
4. Un suceso cambia el estado de un proceso de bloqueado a ejecutable.

Políticas de planificación:

Monoprocesadores

Se pretende obtener buen rendimiento y servicio. Para saber si un proceso tiene un buen servicio definimos para un proceso P con tiempo de ráfaga(~quantum) t:

a) Tiempo de Espera(T): tiempo desde que se remite una solicitud hasta que se produce la respuesta

b) Tiempo de espera(M): tiempo que ha estado un proceso en cola de preparados (T-t)

c) Penalización(P = T/t)

d) Índice de respuesta(R = t/T): tiempo que P está recibiendo servicio.

Otras medidas son tiempo del núcleo(tiempo perdido por el SO tomando decisiones de planificación), tiempo de inactividad (tiempo en el que la cola de ejecutables está vacía) y tiempo de retorno(tiempo necesario para ejecutar un proceso completo).

Clasificación de Políticas de planificación

1. No apropiativas(no expulsivas): Una vez que se asigna la CPU a un proceso no se le puede retirar hasta que éste se finalice o se bloquee
2. Apropiativas(expulsivas): El SO puede apropiarse del procesador cuando lo decida.

Los más importantes conocidos son:

- **FCFS**(First Come First Served). Los procesos se ejecutan según llegan a la cola de ejecutables. Es **no apropiativo**, cada proceso se ejecuta hasta que finalice o se bloquee. Es fácil de implementar pero pobre de prestaciones y todos los procesos pierden el mismo tiempo esperando en la cola de ejecutables. De esta forma, los procesos cortos se penalizan muchos y los largos poco.

- **SJF**(Shortest Job First). Se selecciona el proceso que requiere menos tiempo de CPU. Si hay dos procesos que tengan las mismas condiciones se sigue FCFS. Es **no apropiativo**. Se necesita saber el tiempo de ejecución estimado y disminiye el tiempo de espera para procesos cortos, los largos se ven afectados. El tiempo medio de espera es bajo.
- **SRTF**(El más corto primero apropiativo). Si un proceso entra a la cola de ejecutables se comprueba si su tiempo de servicio es menor que el tiempo de servicio restante al proceso ejecutándose. Si es menor, se cambia de contexto y se ejecuta y si no sigue el que estaba ejecutándose. El tiempo de respuesta es menor excepto para procesos largos y hay menos penalización en promedio.
- **Planificación por prioridades**. Se asocia a cada proceso un entero de prioridad. A la CPU se le asigna el proceso con mayor prioridad(enteros menores es más prioridad). Puede ser apropiativa o no apropiativa. El **problema** es que los procesos de baja prioridad pueden no ejecutarse nunca, pero como **solución** con el paso del tiempo se incrementa la prioridad de los procesos.
- **Round-Robin** (Por turnos). La CPU se asigna a los procesos en intervalos de tiempo(QUANTUMS). Si el proceso finaliza o se bloquea antes de agotar el quantum, se libera la CPU y se toma el siguiente proceso de la cola FIFO de ejecutables y se le asigna un quantum completo. Si el proceso no termina en su quantum, se interrumpe y se coloca al final de la cola de ejecutables (Es por tanto apropiativo). Este sistema penaliza a todos los procesos de la misma forma. Y las ráfagas muy cortas están más penalizadas. Si el valor del quantum es muy grande se convertirá en FCFS y si es muy pequeño el sistema monopoliza la CPU por los cambios de contexto.
- **Colas múltiples**. La cola de preparados se divide en varias colas y cada proceso se asigna permanentemente a una cola. Cada cola puede tener su algoritmo de planificación, pero debe existir una planificación entre colas (planificación con prioridades fijas o tiempo compartido).
- **Colas múltiples con realimentación**. Los procesos se pueden mover entre colas, pero requiere que definamos el número de colas, qué algoritmo de planificación hay en cada cola,un método para saber cuándo hay que cambiar un proceso de cola, otro para determinar en qué cola se introducirá un proceso y un algorimo de planificación entre colas. Sin embargo, mide el tiempo de ejecución del comportamiento real de los procesos. Es la más general, se usa en Unix, Linux, Windows...

Planificación en multiprocesadores

Se estudian tres aspectos relacionados: Asignación de procesos a procesadores(Cola para cada procesador y cola para todos los procesadores), uso de multiprogramación en cada procesador y activación del proceso.

- Planificación de procesos: Igual que en monoprocesadores pero se tiene en cuenta el número de CPUs y la asignación y liberación de proceso y procesador.
- Planificación de Hilos: para explotar el paralelismo dentro de una aplicación. En multiprocesadores existen varias formas.

a) Compartición de carga: Hay una cola global de hilos preparados y si un procesador está muy ocioso, se selecciona un hilo de la cola.

b) Planificación en pandilla: Se planifican varios hilos de un mismo proceso para ejecutarse dentro de un conjunto de procesadores al mismo tiempo. Es útil cuando los hilos necesitan sincronizarse.

c) Asignación de procesador dedicado: Se asigna un procesador a cada hilo hasta que termine la aplicación, pero puede que ocurra que haya procesadores vacíos(no hay multiprogramación).

d) Planificación dinámica: La aplicación permite que varíe dinámicamente el número de hilos de un proceso y el SO ajusta la carga para usar mejor los procesadores.

- Planificación de sistemas de tiempo real. Se enfoca según cuándo el sistema realice un análisis de viabilidad de la planificación (si puede atender a todos los eventos en su tiempo), si se realiza estática o dinámicamente o si el resultado del análisis produce un plan de planificación o no. Se utilizan enfoques estáticos dirigidos por una tabla(planificación que determina cuándo empezará cada tarea), estáticos expulsivos dirigidos por prioridad (sólo da prioridad a las tareas, no genera una planificación), enfoques dinámicos basados en plan(determina la viabilidad en tiempo de ejecución y se acepta si se pueden satisfacer sus restricciones de tiempo) y enfoques dinámicos de menor esfuerzo(sin análisis de viabilidad, se intenta cumplir los plazos y si no se cumple se aborta el proceso).

Problema de inversión de prioridad

Este problema ocurre en los planificadores con prioridad cuando una tarea de mayor prioridad espera por una tarea de menor prioridad porque hay un recurso bloqueado de uso exclusivo.

Para evitar el problema, se puede:

1. Establecer una herencia de prioridad (la tarea menos prioritaria hereda la prioridad de la más prioritaria)
2. Establecer un techo de prioridad (al proceso que se le asigna el recurso de uso exclusivo se le da una prioridad más alta)

En ambos, la menos prioritaria vuelve a tener el valor de prioridad que tenía cuando libere el recurso.

- Diseño e implementación de procesos e hilos en Linux

Nos basamos en el kernel 2.6 de Linux.

1. El núcleo identifica a los procesos por su PID
2. En Linux, proceso es la entidad que se crea con la llamada al sistema *fork* (excepto el proceso 0) y clone.
3. Procesos especiales que existen durante la vida del sistema; Proceso 0 (creado "a mano" cuando arranca el sistema, crea al proceso 1), Proceso 1 (Init, antecesor de cualquier proceso del sistema).

Linux: estructura task.

El kernel almacena la lista de procesos como una lista circular doblemente enlazada (task list). Cada elemento es un descriptor de un proceso (PCB) definido en

```
struct task_struct { /// del kernel 2.6.24
 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
 /*...*/
/* Informacion para planificacion */
 int prio, static_prio, normal_prio;
 struct list_head run_list;
 const struct sched_class *sched_class;
 struct sched_entity se;
/*...*/
 unsigned int policy;
 cpumask_t cpus_allowed;
 unsigned int time_slice;
/*...*/
//* Memoria asociada a la tarea */
 struct mm_struct *mm, *active_mm;
/*...*/
 pid_t pid;
/* Relaciones entre task_struct */
 struct task_struct *parent; /* parent process */
 struct list_head children; /* list of my children */
 struct list_head sibling; /* linkage in my parent's children list */
/* Informacion para planificacion y señales */
 unsigned int rt_priority;
 sigset_t blocked, real_blocked;
 sigset_t saved_sigmask; /* To be restored with TIF_RESTORE_SIGMASK */
 struct sigpending pending;
/*...*/
```

Estados de un proceso en Linux

La variable state de task_estruct especifica el estado actual de un proceso.

1. **Ejecucion** (TASK_RUNNING): Se corresponde con dos: ejecutándose o preparado para ejecutarse (en la cola de procesos preparados).
2. **Interrumpible** (TASK_INTERRUPTIBLE): El proceso está bloqueado y sale de este estado cuando ocurre el suceso por el cual está bloqueado o porque le llegue una señal.
3. **No interrumpible** (TASK_UNINTERRUPTIBLE): El proceso está bloqueado y sólo cambia´ra de estado cuando ocurra el suceso que esta esperando (no acepta señales).
4. **Parado** (TASK_STOPPED): El proceso ha sido detenido y sólo puede reanudarse por la acción de otro proceso (por ejemplo, proceso parado mientras está siendo depurado).
5. (TASK TRACED): El proceso está siendo traceado por otro proceso.
6. **Zombie** (EXIT_ZOMBIE): El proceso ya no existe pero mantiene la entrada de la tabla de procesos hasta que el padre haga un wait (EXIT_DEAD).

Modelo de procesos/hilos en Linux

[Diagrama de estados]{diagrama.png}

El árbol de procesos.

Cada *task_struct* tiene un puntero:

1. A la *task_struct* de su padre: struct task_struct *parent
2. A una lista de hijos (llamada children): struct list_head children
3. A una lista de hermanos (llamada sibling): struct list_head sibling

[Arbol de procesos]{diagrama2.png}

Implementación de hilos en Linux

Desde el punto de vista del kernel no hay distincion entre hebra y proceso. Linux implementa el concepto de hebra como un proceso sin mas, que simplemente comparte recursos con otros procesos. Cada hebra tiene su propia *task_struct*. La llamada al sistema *clone* crea un nuevo proceso o hebra.

```
#include <sched.h>
 int clone (int (*fn) (void *), void *child_stack, int flags, void *arg);
```

Hebras Kernel

Aveces es util que el kernel realice operaciones en segundo plano, para lo cual se crean hebras kernel. Las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL). Se ehecutan únicamente en el espacio del kernel. Son planificadas y pueden ser expropiadas. Se crean por el kernel al levantar el sistema, mediante una llamada a clone(). Terminan cuando realizan una operacion do_exit o cuando otra parte del kernel provoca su finalización.

Creación de procesos.

`fork() → clone() → do_fork() → copy_process()`

- Actuación de *copy_process*:

1. Crea la estructura *thread_info (pila Kernel)* y la *task_struct* para el nuevo proceso con los valores de la tarea actual.
2. Para los elementos de *task_struct* del hijo que deban tener valores distintos a los del padre, se les dan los valores iniciales correctos.
3. Se establece el estado del hijo a *TASK_UNINTERRUPTINLE* mientras se realizan las restantes acciones.
4. Se establecen valores adecuados para los *flags* de la `task_struct` del hijo:

- *flag* `PF_SUPERPRIV = 0` (la tarea no usa privilegio de superusuario).
- *flag* `PF_FORKNOEXEC = 1` (el proceso ha hecho *fork* pero no *exec*).

5. Se llama a `alloc_pid()` para asignar un PID a la nueva tarea.
6. Según cuáles sean los *flags* pasados a `clone()`, duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso...
7. Se establece el estado del hijo a `TASK_RUNNING`.
8. Finalmente `copy_process()`termina devolviendo un puntero a la `task_struct` del hijo.

Terminación de un proceso.

- Cuando un proceso termina, el *kernel* libera todos sus recursos y notifica al padre su terminación.
- Normalmente un proceso termina cuando:

1. Se realiza la llamada al sistema `exit()`:
   - De forma explícita: el programador incluyó esa llamada en el código del programa.
   - O de forma implícita: el compilador incluye automáticamente una llamada a `exit()` cuando `main()` termina.
2. Recibe una señal ante la que tiene la acción establecida de terminar.

- El trabajo de liberación lo hace la función `do_exit()` definida en `<linux/kernel/exit.c>`.

Actuación de `do_exit()`:

1. Activa el *flag* `PF_EXITING` del `task_struct` del proceso.
2. Para cada recurso que esté utilizando el proceso, se decrementa el contador correspondiente que indica el número de procesos que lo está utilizando.

- Si vale 0, entonces se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo, si fuera una zona de memoria, se liberaría.

3. El valor que se pasa como argumento a `exit()` se almacena en el campo `exit__code` de `task_struct` (información de terminación para el padre).
4. Se manda una señal al padre indicando la finalización de su hijo.
5. Si aún tiene hijos, se pone como padre de éstos al proceso init (PID=1).

- (dependiendo de las características del grupo de procesos al que pertenezca el proceso, podría ponerse como padre a otro miembro de ese grupo de procesos).

6. Se establece el campo `exit_state` de `task_struct` a `EXIT_ZOMBIE`.
7. Se llama a `schedule()` para que el planificador elija un nuevo proceso a ejecutar.

Puesto que este es el último código que ejecuta un proceso, `do_exit()` nunca retorna.

Planificación de la *CPU* en Linux.

- Planificación modular: clases de planificación,

1. Planificación de tiempo real.
2. Planificación neutra o limpia (*CFS: Completely fair scheduling*).
3. Planificación de la tarea *idle* (no hay trabajo que realizar).

- Cada clase de planificación tiene una prioridad.
- Se usa un algoritmo de planificación entre las clases de planificación por prioridades apropiativo.
- Cada clase de planificación usa una o varias políticas de planificación para gestionar sus procesos.
- La planificación no opera únicamente sobre el concepto de proceso, sino que maneja conceptos más amplios en el sentido de manejar grupos de procesos: entidad de planificación.
- Una entidad de planificación se representa mediante una instancia de la estructura `sched_entity`.

RELACION DE EJERCICIOS TEMAS 1 Y 2

1. Cuestiones sobre procesos, y asignación de CPU:

- ***¿Es necesario que lo último que haga todo proceso antes de finalizar sea una llamada al sistema para finalizar? ¿Sigue siendo esto cierto en sistemas monoprogramados?***

  Si es necesario, puesto que al finalizar un proceso, este debe informar al SO de que su ejecución ha acabado con el fin de que libere el espacio reservado y el PCB de dicho proceso para posteriormente llamar a otro proceso en estado preparado.

  Y en sistemas monoprogramados tambien es necesario puesto que igualmente (y con mas razón) el SO necesita conocer cuando termina el proceso para liberar su espacio de memoria e introducir un nuevo proceso en espera.

- ***Cuando un proceso se bloquea, ¿deberá encargarse él directamente de cambiar el valor de su estado en el descriptor de proceso o PCB?***

  Cuando un proceso se bloquea, el sistema operativo es el encargado de cambiar su estado en el PCB. Al ser datos del sistema operativo, se debe hacer desde el modo kernel, por eso el proceso no

- ***¿Qué debería hacer el planificador a corto plazo cuando es invocado pero no hay ningún proceso en la cola de ejecutables?***

  Este problema es resuelto en muchos sistemas operativos con el proceso NULO que es creado por el sistema en el momento de arranque. El proceso nulo nunca termina, no tiene E/S y tiene la prioridad más baja en el sistema. En consecuencia la cola de listos nunca está vacía, además la ejecución del planificador puede hacerse más rápida al eliminar la necesidad de comprobar si la cola de listos está vacía o no. Algunas de las tareas que se le pueden dar al proceso nulo, por ejemplo, es realizar estadísticas de uso de procesador, o asistencia de vigilancia de la integridad del sistema, etc.

- ***¿Qué algoritmos de planificación quedan descartados para ser utilizados en sistemas de tiempo compartido?***

  Tenemos que usar aquellos algoritmos que favorezcan procesos cortos (ya que estamos en tiempo compartido) generalmente los que no estan basados en quamtum de tiempo. Descartamos FCFS Y el mas corto primero no apropiativo.

2. La representación gráfica del cociente [(tiempo_en_cola_ejecutables + tiempo_de_CPU) / tiempo_de_CPU] frente a tiempo_de_CPU

suele mostrar valores muy altos para ráfagas muy cortas en casi todos los algoritmos de asignación de CPU. ¿Por qué?

Los procesos que usan ráfagas cortas son los procesos cortos, y estos pasan mucho tiempo en cola de ejecutables y muy poco en cpu, entonces como el cociente es (tiempo en cola + tiempo de cpu)/tiempo de cpu pues aumenta mucho para tiempos de cpu muy cortos, en especial si tiempo en cola es grande de por sí. A este cociente se le conoce como penalización

3. Para cada una de las llamadas al sistema siguientes, especificar y explicar si su procesamiento por el sistema operativo requiere la invocación del planificador a corto plazo:

1. **Crear un proceso**
   Depende de si el planificador que estamos usando es apropiativo o no. Si es apropiativo el úlitmo paso al crear un proceso es llamar al planificador,ya que si es apropiativo podria ocurrir que el proceso tuviera que entrear a ejecutarse nada más llegar a la cola de preparados. En caso de el planificador no sea apropiativo no se llama al planificador, puesto que no vamos a tener este problema ya que no se podria retirar la cpu de un proceso que este ejecutandose.

- **Abortar un proceso, es decir, terminarlo forzosamente**
  Si se requiere del planificador a corto plazo, ya que cuando un proceso sale del procesador el planificador a corto plazo debe decidir que proceso de la cola de preparados debe entrar a ejecutarse, sea la politica apropiativa o no apropiativa.

- **Suspender o bloquear un proceso**
  El planificador es el que decide cuando suspender o bloquear un proceso, por lo que es necesario, y luego vuelve a ser necesario para decidir que proceso pasa a ejecutarse.

- **Reanudar un proceso (inversa al caso interior)**
  Al reanudar un proceso, este vuelve a la cola de preparados, luego ocurre lo mismo que en el caso anterior (el caso a)), si es apropiativo habria que llamar al planificador, y en caso de no apropiativo no.

- **Modificar la prioridad de un proceso**
  Lo mismo qu en el caso anterior (el caso a)), ya que si estamos en una politica apropiativa, podria darse el caso de que el proceso deba pasar a ejecutarse por tener mayor prioridad, luego hay que llamar al planificador. En caso de que no fuera apropiativo no hace falta.

4. Sea un sistema multiprogramado que utiliza el algoritmo Por Turnos (Round Robin). Sea S el tiempo que tarda el despachador en cada cambio de contexto. ¿Cuál debe ser el valor de quantum Q para que el porcentaje de uso de la CPU por los procesos de usuario sea del 80%?

Tiempo total de CPU = S+Q

```
T = S+Q
0.8T = Q
--> 5Q/4 - Q = S --> Q = 4S
```

5. Sea un sistema multiprogramado que utiliza el algoritmo Por Turnos (Round-Robin). Sea S el tiempo que tarda el despachador en cada cambio de contexto, y N el número de procesos existente.¿Cuál debe ser el valor de quantum Q para que se asegure que cada proceso "ve" la CPU al menos cada T segundos?

Se ponen las soluciones de ambos grupos porque las dos son válidas

S: tiempo que tarda el despachador en cada cambio de contexto

N: número de procesos existente

Q: valor de quantum

T: segundos

Para N > 1:

$T = N \cdot S + (N-1) \cdot Q \Rightarrow Q = \frac{T-N \cdot S}{N-1}$

Esto es así ya que suponemos que la última vez que el proceso ve la CPU es antes de gastar el último quantum.

Forma alternativa de expresar la frase anterior: consideramos el tiempo T desde que un proceso acaba hasta que vuelve a ser llamado

---

S+Q es el tiempo que tarda en hacer el cambio de contexto y en ejecutarlo, esto se repite para cada proceso. Por lo tanto, para que todo los N procesos pasen por el procesador se tienen N cambios de contextos y ejecuciones N(S+Q) y esto tienen que ocurrir en menos de T segundos para que todos los procesos sean vistos N(S+Q) <= T. Despejando Q tenemos Q <= (T - (N * S))/N

6 ¿Tiene sentido mantener ordenada por prioridades la cola de procesos bloqueados? Si lo tuviera, ¿en qué casos sería útil hacerlo?

No, normalmente no tiene sentido. Puede tener sentido en el caso de que haya varios procesos bloqueados con distintas prioridades esperando al mismo evento.

7. ¿Puede el procesador manejar una interrupción mientras está ejecutando un proceso si la política de planificación que utilizamos es no apropiativa?

Una interrupción debería de ser procesada independientemente del algoritmo de planifiación que se esté utilizando. Si solo tenemos un procesador no puede estar haciendo las dos cosas a la vez. Si llega una interrupcion, se le quitaria la CPU al proceso y se le da a dicha interrupcion y cuando se despache la interrupcion, y volvemos al proceso.

8. Suponga que es responsable de diseñar e implementar un sistema operativo que va a utilizar una política de planificación apropiativa. Suponiendo que tenemos desarrollado el algoritmo de planificación a tal efecto, ¿qué otras partes del sistema operativo habría que modificar para implementar tal sistema? y ¿cuáles serían tales modificaciones?

SO apropiativo: el SO puede apropiarse del procesador cuando lo decida. La planificación apropiativa es útil en los sistemas en los cuales los procesos de alta prioridad requieren una atención rápida.

Para políticas de apropiación basadas en quantum necesitaría aparte un temporizador encargado de llevar los quantum así como el manejo de interrupciones que cambiam de bloqueado a preparado y después llamar al planificador, para tareas no basadas en tiempos de quantum tenemos que cambiar otras partes del sistema operativo que se ven implicadas.

9. En el algoritmo de planificación FCFS, la penalización ((t + to de espera) / t), ¿es creciente, decreciente o constante respecto a t (tiempo de servicio de CPU requerido por un proceso)? Justifique su respuesta.

La constante es creciente, lo ilustraremos con un ejemplo: Si tenemos 3 procesos, P1,P2 y P3 que tardan todos 10 segundos en ejecutarse, vemos que los tiempos de espera

$$\frac{t + te}{t} = 1 + \frac{te}{t}$$

Pero te siempre va a aumentando para cada proceso que se aleje más del primer proceso que se va a ejecuta, lo que implica que $\frac{te}{t}$ va decreciendo al aumentar t, por tanto es decreciente.

10. En la tabla siguiente se describen cinco procesos:**

Proceso	Tiempo de creación	Tiempo de CPU
A	4	1
B	0	5
C	1	4
D	8	3
E	12	2

Si suponemos que tenemos un algoritmo de planificación que utiliza una política FIFO (primero en llegar, primero en ser servido), calcula:**

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

- Tiempo medio de respuesta: 12/5 = 2'4

A = 5 + 1 = 6

B = 0 + 5 = 5

C = 4 + 4 = 8

D = 2 + 3 = 5

E = 1 + 2 = 3

- Tiempo medio de espera: 27/5 = 5'4 A = 9 - 4 = 5

B = 0

C = 5 - 1 =4

D = 10 - 8 = 2

E = 13 - 12 = 1

- La penalización, es decir, el cociente entre el tiempo de respuesta y el tiempo de CPU. A = 6/1 = 6

B = 5/5 = 1

C = 8/4 = 2

D = 5/3

E = 3/2 = 1'5

11. Utilizando los valores de la tabla del problema anterior, calcula los tiempos medios de espera y respuesta para los siguientes algoritmos:

1. **Por Turnos con quantum q=1**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A						X									
B	X		X		X			X			X				
C		X		X			X		X						
D										X		X		X	
E													X		X

Cola:

T0 B, T1 CB, T2 BC, T3 CB, T4 BAC , T5 ACB, T6 CB, T7 BC, T8 CDB, T9 DB, T10 BD, T11 D, T12 ED, T13 DE, T14 E

Tiempo de espera:

MA = 1
MB = 6
MC = 4
MD = 3
ME = 1

M= 15/5 = 3

Tiempo de respuesta:

TA = 1
TB = 0
TC = 0
TD = 1
TE = 0

T = 2/5 = 0,4

- **b) Por Turnos con quantum q=4**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A									X						
B	X	X	X	X						X					
C					X	X	X	X							
D											X	X	X		

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E														X	X

Cola:

T0 B, T1 C, T2 C, T3 C, T4 CAB, T5 AB, T6 AB, T7 AB, T8 ABD, T9 BD, T10 D, T11 , T12 E, T13 E, T14

Tiempo de espera:

MA = 4
MB = 5
MC = 3
MD = 2
ME = 1

M = 15/5 = 3

Tiempo de respuesta:

TA =
TB =
TC =
TD =
TE =

T = ? / 5

- **c) El más corto primero (SJF). Suponga que se estima una ráfaga igual a la real.**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A						X									
B	X	X	X	X	X										
C							X	X	X	X					
D											X	X	X		
E														X	X

Cola:

T0 B, T1 C, T2 C, T3 C, T4 AC, T5 AC, T6 C, T7 , T8 D, T9 D, T10 D, T11 , T12 E, T13 E, T14

Tiempo de espera:

MA = 1
MB = 0
MC = 5

MD = 2
ME = 1

M = 9/5 = 1.8

Tiempo de respuesta:

TA =
TB =
TC =
TD =
TE =

T = ?/5

12 Calcula el tiempo de espera medio para los procesos de la tabla utilizando el algoritmo: el primero más corto apropiativo (o primero el de tiempo restante menor, SRTF).

0	1	2	3	4	...	8	9	...	13	14	...	19	20	...	26
A	B	A	A	C	...	C	D	...	D	E	...	E	C	...	C

El tiempo medio de espera es el tiempo que el proceso pasa en la cola esperando a volver a ejecutarse.

M(A) = 1

M(B) = 0

M(C) = 12

M(D) = 0

M(E) = 2

Luego M_t = 15/5 = 3

13. Utilizando la tabla del ejercicio anterior,

Proceso	Tiempo de creación	Tiempo de CPU
A	0	3
B	1	1
C	3	12
D	9	5

```
E            12              5
```

dibuja el diagrama de ocupación de CPU para el caso de un sistema que utiliza un algoritmo de colas múltiples con realimentación con las siguientes colas:

```
Cola Prioridad Quantum

1       1         1

2       2         2

3       3         4
```

y suponiendo que:

- ***Todos los procesos inicialmente entran en la cola de mayor prioridad (menor valor numérico). Cada cola se gestiona mediante la política Por Turnos.***
- ***la política de planificación entre colas es por prioridades no apropiativo.***
- ***un proceso en la cola i pasa a la cola i+1 si consume un quantum completo sin bloquearse.***
- ***cuando un proceso llega a la cola de menor prioridad, permanece en ella hasta que finalice.***

14. Suponga que debe maximizar la eficiencia de un sistema multiusuario y que está recibiendo quejas de muchos usuarios sobre los pobres tiempos de respuesta (o tiempos de vuelta) de sus procesos. Los resultados obtenidos con una herramienta de monitorización del sistema nos muestran que la CPU se utiliza al 99'9% de su tiempo y que los procesadores de E/S están activos sólo un 10% de su tiempo. ¿Cuales pueden ser las razones de estos tiempos de respuesta pobres y por qué?

1. **El quantum en la planificación Round-Robin es muy pequeño.**

   Un cuantum pequeño genera sobrecarga pero no malos tiempos de respuesta.

- **La memoria principal es insuficiente.**

  Si la memoria principal fuese insuficiente los procesadores de dispositivos estarían más ocupados por ejemplo accediendo a disco

- **El sistema operativo tiene que manejar mucha memoria principal por lo que las rutinas de gestión de memoria están consumiendo todos los ciclos de CPU.**

  La complejidad de la gestión de memoria no tiene que ver el tamaño de la misma.

- **La CPU es muy lenta.**

  Si la CPU fuese muy lenta, hubiésemos obtenido un 100% de ocupación.

- **El quantum en la planificación Round-Robin es muy grande.**

  Por último, un cuantum grande si nos da unos tiempo de respuesta malos y es compatible con los valores mostrados por la herramienta de vigilancia, pues en con un quantum grande todos los procesos largos se ejecutarían ocupando todo el tiempo disponible en el procesador, sin dar cabida a los procesos cortos. Por lo tanto la CPU pasa mucho tiempo ocupada con estos procesos mientras que apenas produce respuesta en la E/S.

15. Compare el rendimiento ofrecido al planificar el conjunto de tareas multi-hebras descrito en la tabla y bajo las siguientes configuraciones:

a) Sistema operativo multiprogramado con hebras de usuario. En este sistema se dispone de una biblioteca para la programación con hebras en el espacio de usuario. El algoritmo de planificación de CPU utilizado por el SO es Round-Robin con un quantum de 50 u.t. (unidades de tiempo). El planificador de la biblioteca de hebras reparte el quantum del proceso (tarea) entre las hebras utilizando Round-Robin con un quantum para cada hebra de 10 u.t. Suponga que no existe coste en el cambio de contexto entre hebras ni entre procesos.

b) Sistema operativo multiprogramado con hebras kernel. El SO planifica las hebras usando Round-Robin con un quantum de 10 u.t. Como en el apartado anterior,suponga que no existe coste en la operación de cambio de contexto. Considere además que las operaciones de E/S de un proceso únicamente bloquean a la hebra que las solicita. Suponga en ambos casos que los dos procesos están disponibles y que el planificador entrega la CPU al proceso P1. Para realizar la comparación represente en cada caso el diagrama de ocupación de CPU y calcule el grado de ocupación de la CPU (tiempo CPU ocupada tiempo total).

Proceso	Hebras	Ráfaga de CPU	Tiempo de E/S	Ráfaga de CPU
	Hebra1	20	30	10
P1	Hebra2	30	-	-
	Hebra3	10	-	-
P 2	Hebra1	30	30	10
	Hebra1	40	-	-

Tiempo	10	20	30	40	50	60	70	80	90	100	110	120	130	140
A	P1.1	P1.2	P1.3	P1.1	P2.1	P2.2	P2.1	P2.2	P2.1	P1.2	P1.1	P1.2	P2.2	P2.1
B	P1.1	P1.2	P1.3	P1.1	P2.2	P1.1	P1.2	P2.1	P2.2	P1.2	P2.1	P1.1	P2.2	P2.2

%CPU de A = 1

%CPU de B = 1

16. ¿EL planificador CFS de Linux favorece a los procesos limitados por E/S (cortos) frente a los limitados por CPU (largos)? Explique cómo lo hace.

Sí. Los procesos cuyo tiempo consumido de CPU es más largo tienen un vruntime superior. El vruntime depende del tiempo real que el proceso ha consumido de CPU, su prioridad y su peso. Como el planificador ejecutará los procesos de menor vruntime se verán beneficiados los procesos más cortos (en este caso E/S).

17 ¿Cuál es el problema que se plantea en Linux cuando un proceso no realiza la llamada al sistema wait para cada uno de sus procesos hijos que han terminado su ejecución? ¿Qué efecto puede producir esto en el sistema?

Si no se realiza la llamada wait para cada uno de sus hijos aunque acabe el proceso padre, el proceso hijo se mantiene en estado zombie, ocupando una entrada en la tabla de procesos y su contexto es descargado de memoria, existiendo aunque no se esté ejecutando. Si un proceso no es esperado por su padre, continúa consumiendo recursos (tabla de entrada, memoria, el planificador lo tiene en cuenta,...), provocando la ineficiencia del sistema. No se podrían crear nuevos procesos por estar la memoria ocupada, se eliminarán únicamente al apagar el ordenador

---

Zombie

EXIT_ZOMBIE

El proceso ya no existe pero mantiene la entrada de la tabla de procesos hasta que el padre haga un wait

(EXIT_DEAD)

18 La orden clone sirve tanto para crear un proceso en Linux como una hebra.

a) Escriba los argumentos que debería tener clone para crear un proceso y una hebra.

b) Dibuje las principales estructuras de datos del kernel que reflejan las diferencias entre ambas.

c)

Para crear un proceso:

```
clone(SIGCHILD, 0)
```

Para crear una hebra:

```
clone(..., CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD, ...)
```

b)

Veamos un dibujo sobre las hebras:

hebras

El dibujo sobre los procesos hijo sería similar, excepto que no se compartiría el espacio de direcciones, ni los archivos abiertos(*), y cada uno tendría su propio PID.

Al comienzo, los archivos abiertos y el espacio de direcciones es el mismo en el hijo que en el padre, pero a partir de ahí cada uno trabaja en su propia copia, sin que los cambios de uno afecten al otro.