

# **SISTEMAS OPERATIVOS**

**2º CURSO EN GRADO DE INGENIERÍA INFORMÁTICA**

**GREGORIO CORPAS PRIETO**



# INDICE

TEMA 1	1
Introducción a los sistemas operativos.....	1
Introducción.....	1
El SO como máquina virtual.....	1
El SO como gestor de recursos.....	1
Planificación y gestión de recursos.....	3
Máquina multinivel.....	4
Lenguaje de Alto nivel.....	4
Ensamblador.....	4
Microprogramación.....	5
Componentes del procesadores.....	6
Registros del procesador.....	6
Ejecución de instrucciones.....	7
Búsqueda y ejecución de instrucción.....	7
Interrupciones.....	7
Salvado de contexto debido a interrupción.....	8
Múltiples interrupciones.....	9
Sistema de Entrada-Salida (E/S).....	11
Técnicas de comunicación de E/S.....	12
E/S Programada.....	12
E/S Dirigida por Interrupciones.....	12
Acceso directo a memoria DMA.....	13
Multiprogramación.....	14
Control de procedimientos (PILA).....	15
Desarrollos impulsores de los SSOO modernos.....	16
Multihilo.....	17
Multiprocesamiento simétrico.....	17
Historia.....	19
Unix tradicional.....	19
Unix modernos.....	19
Linux.....	19
TEMA 2	1
Procesos.....	1
TEMA 3	1
Comunicación entre procesos.....	1
Introducción.....	1
Sección crítica y exclusión mutua.....	2
Algoritmo de Dekker.....	3
Algoritmo de Petterson.....	8
Soporte Hardware para exclusión mutua.....	9
Deshabilitar Interrupciones.....	9
Instrucciones máquina especiales.....	9
Instrucción Test and Set.....	9
Instrucción Exchange.....	10
Semáforos.....	12
Semáforo General.....	12
Semáforo Binario.....	13

Monitores.....	14
Interbloqueo e inanición.....	15
Categorías de recursos.....	16
Paso de mensajes.....	17
Sincronización.....	17
Direccionamiento.....	17
Formato.....	18
Disciplina de Cola.....	19
TEMA 4.....	1
Planificación.....	1
Objetivos.....	1
Criterios para la planificación.....	2
Modos de decisión.....	3
Sin expulsión o apropiativos.....	3
Con expulsión o apropiativos.....	3
Tipos de planificación.....	4
Planificación a largo plazo.....	4
Planificación a medio plazo.....	4
Planificación a corto plazo.....	4
Algoritmos de planificación.....	6
FCFS (First Come First Served).....	6
Round Robin (turno rotatorio).....	6
SPN (Shortest Process Next).....	7
SRT (Shortest Remaining Time).....	7
Retroalimentación.....	8
FSS (Faire Share Scheduling).....	8
Comparación de rendimiento.....	10
Planificación Unix Tradicional.....	11
Planificación multiprocesador.....	11
Granularidad.....	11
Paralelismo independiente.....	11
Paralelismo grano grueso y muy grueso.....	11
Paralelismo grano medio.....	11
Paralelismo grano fino.....	11
Aspecto de diseño.....	12
Asignación de procesos a procesadores.....	12
Uso de multiprogramación en monoprocesadores.....	12
Activación de procesos.....	12
Planificación de hilos.....	13
Compartición de carga.....	13
Planificación en pandilla.....	13
Asignación de procesador dedicado.....	13
Planificación dinámica.....	13

# TEMA 1

## INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

---

### Introducción

El sistema operativo se encarga de **interactuar** directamente con el **hardware**, de modo que controla la ejecución de programas de aplicación y **abstrae** al usuario de la complejidad en el manejo de hardware. Sus objetivos básicos son:

- **Comodidad** para los usuarios: El SO **permite que** un computador **sea sencillo de usar**, haciendo una **imagen de máquina virtual o extendida** para que los usuarios interactúen con ella.
- **Eficiencia**: El SO **gestiona** los **recursos de sistema** de un modo eficaz.

### *El SO como máquina virtual*

La **utilización** directa de **hardware** es **compleja**, especialmente en operaciones de E/S, por tanto, el **objetivo** primordial es crear una **abstracción sencilla** para que el **usuario** pueda **entenderla**.

La máquina virtual entiende **órdenes de nivel superior**. El sistema operativo tiene que proporcionar servicios para:

- **Crear** programas (depuradores, editores, enlazadores)
- **Ejecutar** programas (carga de código en memoria, inicializar E/S)
- **Operar** con **E/S** (comunicar con periféricos a la hora de trabajar con ellos)
- **Manipular** y Controlar **sistema de archivos** (formato de almacenamiento)
- Detectar **errores** (divisiones por cero, violación de permisos)
- **Control** de **acceso** al sistemas (permisos y control de acceso a recursos)
- **Estadísticas** (para prever el uso y conseguir mejor rendimiento)

### *El SO como gestor de recursos*

Se considera al SO como un **programa gestor** de todos los **elementos** del **sistema**. En este caso **la función es proporcionar** en el debido **orden** y de forma **controlada** el **acceso a los recursos** por los que compiten los procesos: acceso a memoria, periféricos,

uso de CPU...

# Planificación y gestión de recursos

Una **responsabilidad clave** en el SO es la **gestión de los recursos** disponibles para planificar su uso por los procesos activos. Debe tenerse en cuenta los 3 factores:

- **Equitatividad:** Se desea que **todos los procesos** que compiten por un recurso **tengan un acceso equitativo** al mismo, sobre todo en trabajos de la misma categoría.
- **Respuesta diferencial:** Se **discriminará entre trabajos** con diferentes clases de trabajos con diferentes requisitos de servicio. Se **debe** asignar y **planificar** los recursos **para satisfacer al conjunto total**, y tomar las decisiones de forma dinámica.
- **Eficiencia:** Se debe **maximizar la productividad**, **minimizar el tiempo** de respuesta y en caso de sistemas de tiempo compartido acomodar a tantos usuarios como sea posible.

El SO mantiene un **número de colas** en las que los **procesos en memoria principal aguardan** su turno para usar algún recurso.

La **cola a corto plazo** se compone de procesos en memoria principal y están listos para ejecutar cuando la CPU lo permita. Es responsabilidad del **planificador a corto plazo o dispatcher**. Se puede asignar el orden a cada proceso en cola usando alguna **política**, como por ejemplo el turno rotatorio o Round Robin.

La cola a **largo plazo** es una lista de trabajos esperando a usar la CPU. El SO añade trabajos **transfiriendo de esta cola a la de corto plazo**, asignándole en este momento una **porción de memoria principal**. El SO debe asegurarse de no sobrecargar la memoria o el tiempo de procesador por exceso de procesos en cola a corto plazo.

También hay una **cola de E/S por cada dispositivo** de E/S. Varios procesos pueden solicitar el uso de un mismo dispositivo E/S.

Si ocurre una **interrupción** el **SO recibe el control del procesador**. Un proceso puede invocar un servicio de sistema mediante una llamada al sistema. En este caso se trata de un **manejador de llamada a sistema (trap)**. Seguidamente, una vez manejada la interrupción o llamada a sistema se invoca al planificador a corto plazo para que seleccione un proceso para su ejecución.

# Máquina multinivel

El **SO** **abstrae al programador** del trabajo hardware a **bajo nivel** (bits, transistores...) y presenta una interfaz amigable ocultando asuntos complejos como interrupciones, administración de memoria, etc..

Esto es una de las funciones del SO, presentar al usuario una **máquina virtual** con la que sea más sencillo trabajar que directamente con el hardware.

Una **máquina virtual** es aquella que **basada en una máquina más elemental presenta más facilidad de uso** incluyendo toda su funcionalidad. Se puede presentar en varias capas, por ejemplo: HARDWARE → Sistema operativo → Aplicaciones

El **lenguaje máquina** está compuesto por la lógica booleana (0 y 1) que son los **valores** que toma un **Bit** (Binary Digit). Las instrucciones que comprenden los computadores están escritas con estos dígitos.

**Para paliar el tedio** que supone **trabajar** directamente **con bits surge** el concepto de **máquina multinivel**, que consiste en una estructuración en capas bajo una serie de abstracciones donde cada capa se apoya en la que está debajo de ella, facilitando el trabajo con el SO.

## Lenguaje de Alto nivel

NIVEL 5	Lenguaje C → Compilación
NIVEL 4	Ensamblador → Lenguaje máquina
NIVEL 3	Sistema operativo
NIVEL 2	Arquitectura de instrucciones (Microprograma)
NIVEL 1	Microinstrucciones
NIVEL 0	Lógica digital

Son los **usados por los programadores** para expresar los programas en un lenguaje más amigable con el pensamiento de este.

**C** es un ejemplo de lenguaje de alto nivel (**nivel 5**).

## Ensamblador

Una vez el código en C se **compila**, es **traducido** a lenguaje **ensamblador (nivel 4)**. Este lenguaje ensamblador.

Este lenguaje aún no lo entiende la máquina. **Se basa en instrucciones básicas** que forman un repertorio que contiene instrucciones máquina (**niveles 3 y 2**).

**Según la arquitectura** del microprocesador y el número de registros, **dichas instrucciones** mnemónicas y la funcionalidad de estas **difieren** de unos a otros.

En el **nivel 3** el sistema operativo reorganiza algunas instrucciones a **nivel de memoria** para posibilitar la **ejecución paralela** de programas.

**El funcionamiento del SO** está **codificado** a nivel de instrucciones, es decir, **en código máquina**. Dicho código es cargado al arrancar nuestra computadora.

A esta parte del código del SO no puede acceder el **usuario normal**, que **como mucho** puede **trabajar** en **niveles 5 o 4**.

El **diseñador del sistema** es el que **puede modificar** a tan **bajo nivel**, diferenciándose así los modos usuario y núcleo.

## Microprogramación

El diseño de procesadores se realiza mediante 2 técnicas:

- **Cableados:** Son **procesadores** cuyo **repertorio** está **codificado directamente sobre el silicio** para un juego de instrucciones concreto. El hardware directamente ejecuta las instrucciones del nivel 2.
- **Microprogramados:** Son procesadores que tienen una **unidad de control genérica** e implementan un **juego de instrucciones** u otro dependiendo de un microprograma software (repertorio programa) en niveles 2 a 1. El **lenguaje máquina no actúa directamente sobre el hardware**, si no que **es interpretado** del nivel 2 al 1 **por** un programa llamado **microprograma**. Este divide una instrucción máquina en varias más sencillas.

A día de hoy la **microprogramación prácticamente** ha **desaparecido**, debido a estos factores:

- Existen **herramientas** avanzadas para **diseñar complejas unidades de control** con un bajísimo porcentaje de error de diseño.
- Las unidades de control **cableadas** son significativamente **más eficientes** que una unidad microprogramada.



# Componentes del procesadores

Al más alto nivel de arquitectura Von Neumann, un computador consta de procesador, memoria y componentes de E/S, todos ellos interconectados mediante un bus de sistema:

- **Procesador:** Controla el funcionamiento del computador y realiza las funciones de proceso de datos. Se denomina CPU.
- **Memoria principal:** Almacena datos y programas. Suele ser volátil. Se le denomina memoria primaria o real.
- **E/S:** Transfieren datos entre el computador y su entorno externo, tales como periféricos de memoria secundaria, equipos de comunicaciones y terminales.
- **Bus de sistema:** Comunica procesadores, memoria y módulos E/S.

## Registros del procesador

El procesador incluye registros que proporcionan un tipo de memoria más rápida que memoria principal, pero de menor capacidad:

- **Registros visibles al usuario:** Permiten al programador en lenguaje máquina o ensamblador **minimizar referencias a memoria** optimizando el uso de registros.
- **Registros de control y estado:** Usados por el procesador para **controlar** su **operación** y por **rutinas** del SO para **controlar** la **ejecución de programas**. Existen varios registros para controlar el funcionamiento del procesador, algunos se puede acceder mediante instrucciones máquina en lo que se denomina **modo kernel**. Ejemplos de estos son:
  - **R DI M (MAR):** Registro de dirección de memoria
  - **R DA M (GPR):** Registro de datos de memoria
  - **R DI E/S:** Registro de dirección de E/S
  - **R DA E/S:** Registro de dato de E/S
  - **PC:** Contador de programa, contiene la dirección de la siguiente instrucción de memoria
  - **IR (OPR):** Contiene la última instrucción leída

La **PSW** o **palabra de estado de sistema** es un **registro o conjunto** de ellos que almacena **información de estado**, bit para habilitar/deshabilitar interrupciones, bit para modo supervisor/usuario.

Uno de estos registros puede ser el que almacena el desbordamiento en una operación aritmética.

# Ejecución de instrucciones

Un **programa** a ejecutar consta de **varias instrucciones** almacenadas **en memoria**. El procesamiento de una instrucción consta de **dos pasos: Fetch** (o fase de búsqueda de instrucción en memoria) y **Ejecución** (o fase ejecución de la instrucción).

En la ejecución del programa se repite esto tantas veces como instrucciones existan, pudiendo cada instrucción involucrar varias operaciones.

La ejecución de un programa se **detiene** cuando se **apaga** la máquina, se produce algún **error** irreparable o se ejecuta una instrucción de detención (**halt**).

## Búsqueda y ejecución de instrucción

El PC tiene la dirección de la siguiente instrucción y se incrementa al ejecutarse una instrucción.

El valor de la instrucción es transferido al registro de instrucción IR, donde será interpretada y llevará a cabo la acción requerida:

- **Transferencia** entre **procesador-memoria** (almacenamientos, cargar de memoria)
- **Transferencia** entre **procesador-E/S** (almacenamientos, cargas)
- **Procesamiento aritmético/lógico de datos** (sumas, restas, comparaciones)
- **Control** (saltos, bifurcaciones)

## Interrupciones

Los **computadores proporcionan** un **mecanismo** para **mejorar** el **uso** del **procesador**, mediante el cual otros módulos como el de E/S **pueden interrumpir** el **secuenciamiento** normal del mismo.

Las interrupciones pueden ser software y hardware:

- **Software:**
  - De **programa**: **Generada** por alguna **condición resultante** al **ejecutar** una **instrucción**, tales como **desbordamientos**, **división por cero** y otras ejecuciones que violan el uso de las órdenes.
  - Por **temporizador**: Generada por un **temporizador** del **procesador**, permitiendo al sistema ejecutar **funciones cíclicamente**.
- **Hardware:**

- De **E/S**: Generada por un controlador E/S para **señalar conclusión** de operación o **indicar** condiciones de **error**.
- Por **fallo de hardware**: Generada por un fallo en el **suministro de energía**, error de **paridad** en memoria, etc.

Partiendo del supuesto de que los dispositivos E/S son mucho más lentos que el procesador, si tenemos un programa que está haciendo constantes operaciones en un dispositivo E/S, el procesador se acomoda a la velocidad de dicho dispositivo, perdiendo así muchos ciclos de tiempo, lo cual resulta improductivo.

**Gracias** a las **interrupciones**, el **procesador** puede **ejecutar** otras **instrucciones** mientras que las operaciones de E/S se llevan a cabo **paralelamente**.

Una vez la **operación** de E/S se **ha llevado a cabo**, el dispositivo estará **listo** para recibir nuevos datos y se lo hará saber al procesador enviando una **señal de petición de interrupción**. El procesador suspenderá la ejecución actual (**salvando su contexto**) y saltará a una **rutina de servicio** (Interrupt Service Routine o **ISR**) específica de **ese dispositivo** de E/S, conocida como **manejador de interrupción** y **reanudando** la ejecución original **tras atender al dispositivo**.

Por **atender** al dispositivo entendemos

- Inspeccionar que la **operación** se hizo con **éxito** o **fracaso**
- Si la **CPU requiere datos del dispositivo**, la ISR tiene que enviarlos a este
- El **dispositivo** puede **requerir** algún tipo de **datos**, dando la ISR paso a la rutina para proceder con ello

Las interrupciones se pueden dar durante el transcurso de un programa, y para el programador, todo el proceso de **salvado y carga de contextos es transparente**, puesto que se encargan de ello la CPU y el SO.

Para usar las interrupciones, se ha añadido una **nueva fase** al **ciclo de instrucción**:

- Fase de **búsqueda**: Busca la siguiente instrucción (fetch)
- Fase de **ejecución**: Se ejecuta la instrucción
- Fase de **interrupción**: **Comprueba** si existe una interrupción, y **en caso afirmativo** se **salva el contexto** y ejecuta el **manejador de interrupción** o ISR. Después se carga el contexto de nuevo y continúa la ejecución del programa.

## Salvado de contexto debido a interrupción

La aparición de una **interrupción dispara** los siguientes **eventos**:

- **Dispositivo E/S genera señal de interrupción** en un bus hacia el procesador, modificando un registro.

- El **procesador** no continúa con la instrucción consecutiva, si no que **pasa por la fase de interrupción** comprobando si hay alguna que atender. Si encuentra alguna **determina de que tipo o dispositivo** es (consultando una tabla o tipos de interrupción, la cual puede estar integrada en hardware). Dependiendo de la arquitectura **habrá una rutina** manejadora de interrupción **genérica o una específica** para cada tipo.
- El **procesador** se preparará para **dar el control** a la rutina **manejadora de interrupción**. El primer paso es **salvar** la información de los registros de estado (palabra de estado de sistema **PSW**) y la próxima orden a ejecutar (Contador de programa **PC**). Según el sistema se almacenará **en la pila o en el BCP** (Bloque de control de proceso)
- El procesador **carga** el **PC** con la dirección de **comienzo** de la **rutina** manejadora de interrupción **ISR**
- Se procede a **leer** la **ISR**, lo cual conlleva las operaciones:
  - El **PC y la PSW** han sido **almacenados en pila o BCP**, más es necesario guardar el contenido de los registros del procesador ya que es posible que los utilice el manejador de interrupción y pise el contenido.
  - Salvados los registros se puede **procesar la interrupción**. Esta rutina incluirá un **estado** de la operación E/S (**éxito o fracaso**) u otro evento distinto (**interrupción software**). También puede implicar **envío/recepción** de datos CPU – E/S
  - Finalizada la rutina se **restauran** los **valores de registro** guardados en la pila (contexto del proceso)
  - Por último se **restaura** la **PSW y PC**
- La **próxima instrucción** será a la que apuntaba el **PC antes de la interrupción** y podrá continuar la ejecución por donde iba. Si el **planificador** del sistema lo cree conveniente, tras la interrupción **se puede cargar otro programa y no el que se interrumpió**.

## Múltiples interrupciones

**Durante** la ejecución de una rutina manejadora de interrupción (**ISR**) es **posible** que se realice **otra interrupción** al sistema. Existen **2 alternativas** para tratar esta situación:

- **Inhabilitar interrupciones** durante el proceso de una interrupción. Al entrar a la rutina ISR se deshabilitarán las interrupciones entrantes. Una vez acabada la rutina ISR se rehabilitarán las interrupciones y se comprobarán si ha habido interrupciones. Si las hubiera se manejarán en estricto orden secuencial. La desventaja de esta alternativa es que no tiene en cuenta la urgencia o prioridad.

- **Definir prioridades** que **permitan interrumpir** a una **ISR de menor rango**. Así una vez se **interrumpe una ISR por otra de mayor prioridad**, la **información** de la misma **se guarda en la pila** y se da **paso** automáticamente a la de **mayor prioridad**. Dichas interrupciones se pueden producir consecutivamente, cediendo siempre el control a la más prioritaria. Terminadas todas se vuelve a cargar en contexto del programa principal.

# Sistema de Entrada-Salida (E/S)

Los ordenadores actuales están basados en **arquitectura Von Neumann**, y constan principalmente de:

- **Procesador**
- **Memoria**
- **E/S**

Para interconectar dichas unidades si usan los **buses**:

- **Datos:** 8/16/32/64 bits
- **Direcciones:** Para conectar **CPU con memoria y E/S**
- **Control:** Para enviar **señales** de comunicación desde CPU a el **resto de unidades**

Con dicha arquitectura requeriremos de un **sistema** que **comunique** la **E/S con el procesador**, descargando a este de trabajo siempre que sea posible. Hablamos del **módulo de E/S o controlador**.

El controlador es un circuito integrado o **chip integrado en placa base**. En la mayoría de casos separado físicamente de CPU y memoria, pero unido a estos mediante buses de sistema. En algunos computadores es el **PUENTE NORTE y PUENTE SUR**.

Dicho chip **transfiere** y **controla** flujo de **datos entre memoria, procesador y periféricos**. Los periféricos se conectan a placa mediante diversos módulos tales como SLOTS PCI, USB, PS2..

Un **módulo** de E/S puede **controlar múltiples periféricos**. El SO trabaja directamente con el módulo de E/S y no con el dispositivo periférico.

Debido a que cada periférico es fabricado por un fabricante distinto, con chips, ordenes y modo de trabajar distinto a otro similar, por tanto para **comunicarse el SO con el dispositivo periférico** necesita un **protocolo** de órdenes y **comunicación** software, lo que se conoce como **driver**.

Las **funciones** del sistema E/S son:

- **Envío** de **comandos** a dispositivos, **recibir** sus **interrupciones** y ocuparse de sus **errores**
- Ofrecer **interfaz sencilla entre dispositivos** y el sistema, incluyendo la **CPU**
- Optimizar la E/S **delegando trabajo en el módulo E/S y dispositivos** para liberar de esa carga y tiempos a la CPU
- Permitir **conexión de nuevos dispositivos E/S**
- Actuar de **buffer entre memoria o CPU y los periféricos**

- **Detección de errores**, tanto mecánicos como eléctricos y comunicarlos a CPU

## Técnicas de comunicación de E/S

### *E/S Programada*

En esta técnica **no se hace uso de interrupciones**, por tanto, **al encontrar** una **operación de E/S**, el procesador ejecuta esa acción y **delega al módulo E/S** dicho trabajo.

Luego el **procesador revisa periódicamente** el **estado** del **módulo E/S** hasta que encuentre que se finalizó la operación.

En este caso el procesador no es interrumpido, sino que el es el que interrumpe al módulo para preguntar cíclicamente.

El **juego de instrucciones** de una operación E/S a través de un módulo E/S incluyen las categorías:

- **Control**: Usadas para activar el dispositivo e **indicarle que hacer**, por ejemplo, rebobinar o avanzar un registro.
- **Estado**: Usadas para **comprobar condiciones de estado** asociadas al módulo E/S y sus periféricos
- **Transferencia**: Usadas para leer desde un dispositivo o para **enviar/recibir datos** en el mismo.

### *E/S Dirigida por Interrupciones*

En **esta técnica**, el **procesador** trabaja **paralelamente al módulo de E/S** en lugar de esperar a que se complete el trabajo de este.

Al ejecutar una orden que implique E/S, el **procesador ejecuta** un **mandato al módulo de E/S** y **salva** el **contexto** del proceso **actual** para comenzar a ejecutar otro (normalmente, el proceso actual requiere de E/S, por eso es que se salva su contexto y se sigue con otro). Esto es **decisión del planificador**.

No se debe confundir esto con una interrupción, puesto que la interrupción siempre va en dirección Periférico → CPU y no al revés.

Una vez el **dispositivo de E/S** comienza a trabajar, tarde o temprano **enviará interrupciones a CPU** para informar de como va el trabajo. La **CPU** al acabar cada instrucción **pasa por su fase de interrupciones** y **busca** por si hay alguna, y en este caso, la **CPU salvará** el **contexto** del proceso, la **PSW y el PC** y ejecutará el manejador de interrupción **ISR** y se tramite la interrupción.

En resumen, **la E/S dirigida por interrupciones es más eficiente** que la programada porque elimina la espera innecesaria, aunque **consume más tiempo de procesador** debido a que cada palabra de datos que va de memoria a módulo E/S y viceversa pasa por CPU.

## ***Acceso directo a memoria DMA***

La E/S programada causa espera activa y la E/S por interrupciones aun siendo más eficiente, consume mucho tiempo de procesador. Por este motivo:

- La tasa de transferencia E/S se limita por la velocidad con la que el procesador comprueba el estado del dispositivo
- El procesador está involucrado en la gestión de cada transferencia E/S, se deben ejecutar varias instrucciones por cada una.

Cuando se **transfieren grandes volúmenes** de datos, como en la grabación de un CD, se requiere una **técnica más eficiente**.

El acceso directo a memoria la puede llevar un **módulo separado, conectado al bus de sistema**, o puede **incluirse** en el **módulo E/S**. Cuando el procesador desea leer o escribir, genera un mandato al módulo DMA informando de:

- Si es **lectura o escritura**
- La **dirección del dispositivo** E/S involucrado
- La **posición inicial** desde donde leer o escribir
- **Número de palabras** a leer o escribir

Si el planificador decide continuar con otro proceso, se salvará el contexto del actual, o en caso contrario se continuará.

Mientras tanto, el módulo **DMA transferirá el bloque completo** de datos, palabra a palabra hacia memoria o desde ella, **sin atravesar procesador**, de forma que no interrumpirá constantemente, aprovechando así la capacidad de la CPU.

Con esta técnica el **procesador solo se involucra al principio y al final**, donde una interrupción avisará al procesador y este ejecutará la ISR correspondiente (previo salvado de contexto, PSW y PC), tras la cual el planificador restaurará o escogerá otro proceso en espera.



# Multiprogramación

Es un concepto **antagónico** a la **monoprogramación**, que ejecutaba un único programa y cuando existía una instrucción de E/S permanecía a la espera de la conclusión de esta para continuar el proceso.

La **multiprogramación** corrige esta ineficiencia **asignando** el uso del **procesador a un proceso** distinto al actual **mientras** que se **esté esperando** a una operación de **E/S**, pudiéndose albergar varios programas multiplexados por división en el tiempo.

Cuando la CPU trata con varios programas residentes en memoria la secuencia en la que se ejecutarán dependerá de la prioridad y de la política de planificación.

# Control de procedimientos (PILA)

Una **técnica habitual** para controlar la **ejecución** de **llamadas a procedimiento** y retornos de los mismos es la pila.

La **pila** es un **conjunto ordenado de elementos**, de modo que solo se puede **acceder** a uno de ellos (el que está en **la cima**). El número de **elementos** de la pila **es variable**, por tanto esta **lista** de apilamiento es conocida como **LIFO** (Last In First Out)

La implementación de esta requiere que exista una **reserva de posiciones** dedicados para ella, por tanto, en memoria principal se reservan un bloque de direcciones contiguas. Generalmente el bloque está parcialmente lleno con elementos de pila y el resto estará disponible para su crecimiento.

Se requieren **3 direcciones** para su funcionamiento apropiado, habitualmente guardados en registros de procesador y en el BCP:

- **Puntero a pila:** Contiene la **dirección** de la **cima**. Si se añade un elemento se decrementa su valor, si se extrae un elemento se incrementa su valor.
- **Base de pila:** Contiene la dirección del **primer elemento** en la base de la pila.
- **Límite de pila:** Contiene la dirección del **extremo superior** reservado para pila.

Cuando se hace un salvado de contexto, se **almacena** en la pila la **dirección de retorno** del proceso **llamador**, y **al** ejecutar el **retorno**, se utiliza la **dirección de la cima** (último proceso apilado) **para continuar el programa** donde se quedó.

En ocasiones se pasan **parámetros en las llamadas** a procedimientos, almacenándolos en la memoria justo tras la instrucción de llamada. Después de los parámetros se almacenará el retorno.

Al ejecutarse una **llamada a función** se **apila la dirección de retorno y los parámetros**.

Al retornar se pueden mantener los valores de parámetros. El conjunto de **parámetros y dirección de retorno** se denomina **marco de pila**.

## **Desarrollos impulsores de los SSOO modernos**

- Evolución del hardware
- Nuevas aplicaciones
- Nuevas amenazas de seguridad
- Incremento en velocidad de cómputo (multiprocesadores)
- Nuevos dispositivos de alta velocidades
- Nuevos dispositivos de almacenamientos
- Acceso a internet
- Computación cliente-servidor
- Amenazas de seguridad en Internet por ataques (gusanos, virus, hacking..)

# Multihilo

El **multithreading** es una **técnica** en la que un **proceso** se **divide en** una serie de **hilos** o procesos ligeros que se **pueden ejecutar en paralelo**, distinguiéndose así:

- Thread o **hilo**: Es una **unidad de trabajo**, con **contexto propio de procesador y propia área de pila**. Se ejecuta secuencialmente y se puede interrumpir.
- **Proceso**: Es una **colección de uno o más hilos y recursos de sistema** asociados (memoria, datos, ficheros abiertos..). Esto corresponde al concepto de programa en ejecución. Dividiendo la aplicación en hilos el programador tiene control sobre la modularidad de aplicaciones y temporización de los eventos.

El **multithreading** es útil para **aplicaciones** que ejecutan **tareas independientes**, de modo que se puedan **ejecutar concurrentemente**.

## Multiprocesamiento simétrico

Debido al surgimiento de computadores con **múltiples procesadores** se ideó el **multiprocesamiento simétrico (SMP)**.

Es un término **referido** a la **arquitectura hardware** del computador y al **comportamiento del SO** que convive con dicha arquitectura.

Un multiprocesador simétrico cuenta con las siguientes características:

- Tiene **múltiples procesadores**
- Dichos procesadores **comparten utilidades de memoria principal y E/S** interconectadas por bus de comunicación u otro esquema de conexión interno.
- Todos los procesadores pueden realizar las **mismas funciones (simetría)**

El SO de un SMP **planifica procesos** a través de **todos los procesadores**. Tiene ventajas potenciales frente arquitecturas monoprocesador:

- **Rendimiento**: Si el trabajo se **puede organizar en paralelo**, el sistema alcanzará **mayor rendimiento** que con un monoprocesador. Se reparten los procesos entre los distintos procesadores y se ejecutan concurrentemente.
- **Disponibilidad**: En el SMP, al poderse llevar a cabo las mismas funciones por cada procesador, el **fallo de un procesador** no para la máquina, **solo bajará el rendimiento**.
- **Crecimiento incremental**: Un usuario puede **mejorar el rendimiento** de un sistema **añadiendo un procesador** adicional.
- **Escalado**: Los fabricantes pueden ofrecer productos con **distintos precios y características** basadas en el **número de procesadores** configurado en el

sistema.

Estas características son **beneficios potenciales, no garantizados**. El sistema debe contar con herramientas para explotar dicho paralelismo en el sistema SMP.

La técnica **multithreading y SMP frecuentemente se analizan juntas**, aunque **son utilidades independientes**, ya que un monoprocesador puede usar técnica multithreading para estructurar aplicaciones y procesos de núcleo.

Una máquina **SMP** es **útil** para **procesos** que **no** tienen **hilos**, así, dichos procesos se pueden ejecutar en paralelo.

Ambas utilidades se complementan y usan en conjunto de modo efectivo.

En SMP la existencia de **múltiples procesadores** es un hecho **transparente al usuario**, el SO se encarga de planificar y sincronizar los hilos o procesos en los distintos procesadores.

# Historia

## ***Unix tradicional***

Unix se desarrolló en los laboratorios Bell en los 70.

Unix está influenciado tanto por CTSS como por muchas ideas de Multics, de hecho se dice que es una versión recortada de este último.

El primer hito más notable fue cuando portaron UNIX de la computadora PDP-7 a la PDP-11 (computador de 18 bits a otro más avanzado con arquitectura Von Neumann).

Más tarde fue escrito en lenguaje C. Esto fue algo sin precedentes, ya que hasta entonces se pensaba que un SO debía estar escrito en ensamblador. Hoy prácticamente todas las implementaciones Unix están escritas en C.

En el 74 Unix se describió en una revista técnica, lo cuál despertó gran interés en el sistema y, pronto, universidades e instituciones comerciales obtuvieron licencias.

La primera versión fuera de laboratorios Bell fue la 6, en el 76.

La siguiente versión fue la 7 en el 78, antecesora de los sistemas Unix más modernos.

Unix BSD (Berkeley Software Distribution) se desarrolló en la universidad de California en Berkeley.

## ***Unix modernos***

CONTINUA...

## ***Linux***

Comenzó como una variante Unix para el PC de IBM. Linus Torvalds escribió la versión inicial y la distribuyó por internet sobre el 1991.

Es un SO libre y su código fuente está disponible.

Su éxito se debe a:

- Disponibilidad de paquetes de software libre bajo auspicios de la Free Software Foundation
- Proyecto GNU, iniciado por Stallman en el 83 con el objetivo de crear un SO totalmente libre. Torvalds combinó las herramientas GNU con su núcleo linux y creó el sistema GNU/LINUX

# TEMA 2

## PROCESOS

---

# TEMA 3

## COMUNICACIÓN ENTRE PROCESOS

---

### Introducción

En sistemas multitarea está permitida la coexistencia de varios procesos a la par, por tanto existen 2 modelos de computadora que permiten la concurrencia:

- **Multiprogramación monoprocesador:** Todos los procesos corren sobre un único procesador. El dispatcher del sistema operativo asigna las rodajas de tiempo a cada proceso, intercalando la ejecución de estos, esto es multiplexación por división en el tiempo.
- **Multiprocesador:** Es una máquina con un conjunto de procesadores que comparten memoria principal. Se utilizan técnicas de intercalado (multiplexación por división en el tiempo) y concurrencia en paralelo, por tanto en un instante pueden estar corriendo simultáneamente tantos procesos como procesadores existan.

En ambos modelos existen problemas debido a 2 tipos de interacciones:

- **Acceso a recurso:** Los procesos comparten o compiten por acceder a un recurso físico (acceder al disco duro) o lógico (acceder a una variable). El sistema operativo ha de gestionar este acceso ordenadamente de modo que todos puedan operar sin conflictos.
- **Comunicación y sincronización:** Cuando varios procesos interactúan entre ellos para alcanzar un objetivo común (proceso compilador y ensamblador trabajando sincronizadamente)



# Sección crítica y exclusión mutua

La situación dada cuando 2 o más procesos acceden a un mismo recurso en cuyo caso, el resultado de la manipulación dependerá de quién y cuando ha accedido a dicho recurso, se denomina **condición de carrera**.

Por ello debemos prohibir el acceso simultáneo a lectura y escritura, acotando lo que se conoce como **región o sección crítica**.

El concepto de **exclusión mutua** asegura que si un proceso está accediendo al recurso, bien sea para leerlo o modificarlo, ningún otro podrá hacerlo concurrentemente.

En todo proceso de exclusión mutua ha de existir:

- Recurso al que acceder
- Código predecesor y antecesor a la sección crítica no involucrado en el recurso
- Funciones de acceso y salida a la sección crítica

Para conseguir una buena solución, suponiendo que las instrucciones en lenguaje máquina son atómicas (se ejecutan completa y secuencialmente), hemos de cumplir 4 normas:

- **Exclusión mutua:** Solo un proceso a la par en la sección crítica
- **Independencia hardware:** No se puede especular con la posibilidad de en que procesador correrá cada proceso.
- **Evitar interbloqueo:** Ningún proceso fuera de su sección crítica puede bloquear otro
- **Evitar inanición:** Todos los procesos deben poder acceder al recurso, no se permite una espera eterna. El acceso a sección crítica sucede durante un tiempo finito.

La **coherencia de datos** es otro problema a evitar. Debido a que las instrucciones se intercalan entre ellas, es posible que en cada ejecución los valores resultantes sean distintos si unos valores dependen de otros.

# Algoritmo de Dekker

El primero en implementar el algoritmo de Dekker fue Edsger Dijkstra, creador del semáforo para coordinar procesadores y programas y conocido detractor de la sentencia GOTO.

## ► Primera tentativa:

<pre>/* PROCESS 0 */ • • while (turno !=0)     /* no hacer nada */;     /* sección crítica*/;     turno = 1; •</pre>	<pre>/* PROCESS 1 */ • • while (turno !=1)     /* no hacer nada */;     /* sección crítica*/;     turno = 0; •</pre>
--	--

- En este caso tenemos espera activa mientras la **bandera turno** no de paso a nuestro proceso.
- Una vez se nos da paso ejecutamos la sección crítica y pasamos el turno al otro proceso

### \* Problemas

- ☐ Los procesos lentos retrasan a los procesos rápidos
- ☐ Si cae uno se cae el sistema

► **Segunda tentativa:**

<pre>/* PROCESS 0 */ • • while (estado[1])     /* no hacer nada */; estado[0] = true /*sección crítica*/; estado[0] = false; •</pre>	<pre>/* PROCESS 1 */ • • while (estado[0])     /* no hacer nada */; estado[1] = true /*sección crítica*/; estado[1] = false; •</pre>
--	--

- En este caso tenemos un **vector** llamado **estado** que retorna el estado del proceso pasado por índice (verdadero o falso)
- En un proceso existirá espera activa mientras sea cierto el estado del otro proceso competidor
- Una vez el otro proceso competidor ha terminado su sección crítica, pondrá su estado a falso
- Así el primer proceso podrá entrar, e inmediatamente pondrá su estado a true para bloquear a el proceso competidor.
- Realizará su sección crítica y acto seguido cambiará su estado a false.

\* **Problemas**

- ☐ Al estar ambos inicializados a falso pueden entrar ambos a la vez a la sección crítica (ya que uno depende del estado del otro)
- ☐ Esta solución no es independiente de las velocidades de ejecución de procesos (no se puede dar por hecho que ambos no intentarán acceder simultáneamente al recurso)

► Tercera tentativa:

<pre>/* PROCESS 0 */ • • estado[0] = true; while (estado[1])     /* no hacer nada */; /* sección crítica */; estado[0] = false; •</pre>	<pre>/* PROCESS 1 */ • • estado[1] = true; while (estado[0])     /* no hacer nada */; /* sección crítica */; estado[1] = false; •</pre>
---	---

- Partiendo de la idea anterior inicializamos el valor de estado antes de la espera activa
- En la espera activa aguardamos a que el estado del otro proceso tome valor falso, para así poder dar paso a la sección crítica
- Una vez acabada la sección crítica se indica cambiando el estado a falso

\* Problemas

- ☐ Se garantiza la exclusión mutua, ya que se inicializan los estados antes de entrar a la espera activa, pero si ambos estados se inicializan a la par se causa interbloqueo y ninguno entrará jamás.

► **Cuarta tentativa:**

<pre>/* PROCESS 0 */ • • estado[0] = true; while (estado[1]) {     estado[0] = false;     /*retraso */;     estado [0] = true; } /*sección crítica*/; estado[0] = false; •</pre>	<pre>/* PROCESS 1 */ • • estado[1] = true; while (estado[0]) {     estado[1] = false;     /*retraso */;     estado [1] = true; } /*sección crítica*/; estado[1] = false; •</pre>
--	--

- Se establece el estado del proceso a verdadero
- Se pasa a espera activa mientras el otro proceso no sea falso, por tanto el estado del proceso actual vuelve a ser falso.
- Se espera durante un tiempo de latencia y actualiza el valor de estado del proceso actual a verdadero.
- Se repite el acceso a espera activa, si esta vez el estado del otro proceso es falso podremos entrar a la sección crítica
- Se pone a false para indicar fin de uso del recurso

\* **Problemas**

- ☐ Se intenta arreglar el problema del interbloqueo pero se puede dar el caso de que un proceso ceda el derecho al otro y viceversa formando un círculo vicioso.

## ► Solución válida

- Para poner remedio a la circunstancia de “cortesía mutua” se añade una variable llamada **turno**, la cual indica que proceso tiene preferencia a la hora de insistir y así establecer una brecha y no estén constantemente cediendo el paso. Es una combinación de la tentativa 1 y 4.

```
boolean estado[2];  
int turno;  
void P0()  
{  
    while (true)  
    {  
        estado[0] = true;  
        while (estado[1]  
            if (turno == 1)  
            {  
                estado[0] = false;  
                while (turno == 1)  
                /* no hacer nada */;  
                estado[0] = true;  
            }  
            /* sección crítica */;  
            turno = 1;  
            estado[0] = false;  
            /* resto */;  
        }  
    }  
}  
void P1()  
{  
    while (true)  
    {  
        estado[1] = true;  
        while (estado[0])  
        if (turno == 0)  
        {  
            estado[1] = false;  
            while (turno == 0)  
            /* no hacer nada */;  
            estado[1] = true;  
        }  
        /* sección crítica */;  
        turno = 0;  
        estado[1] = false;  
        /* remainder */;  
    }  
}  
void main ()  
{  
    estado[0] = false;  
    estado[1] = false;  
    turno = 1;  
    paralelos (P0, P1);  
}
```

*//Se declara un vector booleano que contendrá el estado del proceso 0 y 1  
//Se define la variable turno que indica que proceso tiene prioridad para insistir  
  
//Se pone el estado del proceso a verdadero  
//Mientras el otro proceso se ejecute...  
//Si el turno es el de el otro proceso  
  
//Estado de proceso actual falso  
//Mientras sea el turno del otro proceso...  
//Espera activa  
//Estado del proceso actual a verdadero  
  
/\*En el momento en el que el turno no sea de 1 sino nuestro, como el estado del proceso actual entra iniciado a true tanto al invocar a la funcion como al salir del while anterior, tenemos acceso a seccion critica\*/  
  
//Ejecutada la seccion crítica cedemos el turno al otro proceso  
//Nuestro estado de ejecución ya vale falso  
//Fin de proceso 0  
  
//IDEM  
  
//Inicializaciones de estados a false en ambos procesos  
  
//Inicializacion del turno al proceso 1  
//Lanzamiento de hilos en paralelo*

# Algoritmo de Petterson

Es una simplificación del algoritmo de dekker.

## ► Algoritmo:

```
boolean estado[2];           //Se declara un vector de booleanos para el estado
int turno;                   //Se declara un valor entero indicador de turno
void P0( )
{
    while (true)              //Se ejecuta indefinidamente
    {
        estado[0] = true;     //Estado del proceso vale verdadero
        turno = 1;            //Actualizamos turno al otro proceso
        while (estado [1] && turno == 1) //Mientras el otro proceso este activo y sea su turno...
            /* no hacer nada */; //Espera activa
        /* sección crítica */; //Una vez el otro proceso acaba su región crítica,
        estado [0] = false;     //modifica su valor de estado a falso dejando ejecutar a p0,
        /* resto */;            //aparte de si el otro proceso intenta ejecutarse de nuevo,
                                //también actualiza el turno con valor de proceso 0.
                                //Así garantizamos alternancia*/
    }                          //Se cambia el valor del estado a falso
}
void P1( )
{
    while (true)
    {
        estado [1] = true;     //IDEM
        turno = 0;
        while (estado [0] && turno == 0)
            /* no hacer nada */;
        /* sección crítica */;
        estado [1] = false;
        /* resto */
    }
}
void main( )
{
    estado [0] = false;         //Estado de proceso 0 falso
    estado [1] = false;         //Estado de proceso 1 falso
    paralelos (P0, P1);         //Lanzamiento de hilos paralelos
}
```

# Soporte Hardware para exclusión mutua

## *Deshabilitar Interrupciones*

Las máquinas **monoprocesador no pueden ejecutar procesos** en paralelo (**concurrentemente**), solo pueden entrelazarlos (multiplexación por división en el tiempo). Los procesos se ejecutan hasta que son interrumpidos o se invoque un servicio de sistema.

Por este motivo, **para garantizar la exclusión mutua, se deshabilitan las interrupciones a la entrada de la sección crítica para luego rehabilitarlas a su salida.**

Así **se ejecutará la sección crítica de una manera atómica**, en detrimento de la eficiencia de ejecución ya que limitamos la capacidad del procesador de entrelazar programas.

Además, **en sistemas multiprocesador** esta técnica no sirve, debido a que si que **existe concurrencia** entre procesos, **no garantizándose la exclusión mutua.**

## *Instrucciones máquina especiales*

En sistemas multiprocesador, los procesadores comparten acceso a memoria principal común. Los procesadores se comportan independientemente en una relación de igualdad, sin mecanismos de interrupción entre procesadores que garanticen la exclusión mutua.

A nivel hardware, el acceso a memoria excluye cualquier otra petición de acceso a la misma posición. Basándose en este mecanismo, los diseñadores de procesadores proponen varias instrucciones máquina para llevar a cabo 2 acciones atómicas:

- Leer y escribir
- Escribir y comprobar

Durante la ejecución de la instrucción queda bloqueado todo intento de acceso externo a dicha posición y estas acciones se realizan en un único ciclo de instrucción.

## *Instrucción Test and Set*

Es la instrucción comprueba y establece.

- Comprueba el valor del argumento i
- Si vale 0 lo reemplaza por 1 y devuelve verdadero
- Si vale 1 no lo modifica y devuelve falso



<pre> /* programa exclusión mutua */ const int n = /* número de procesos */; int cerrojo; void P(int i) {     while (true)     {         while (!testset (cerrojo))             /* no hacer nada */;         /* sección crítica */;         cerrojo = 0;         /* resto */     } } void main() {     cerrojo = 0;     paralelos (P(1), P(2), . . . ,P(n)); } </pre>	<pre> //Se declara un número n de procesos //Se declara una variable entera compartida llamada cerrojo  //Para cada ejecución de proceso:  //Mientras testset de cerrojo devuelva false... // (lo cual implica que cerrojo vale 1 ya que está activado // porque un proceso está en su sección crítica) //Espera activa  //Una vez algún proceso haya salido de su sección crítica // seteando cerrojo a 0, hay vía libre para pasar a la sección // crítica.  //Finalizada la sección crítica cerrojo vuelve a quedarse // abierto con valor 0  //En la ejecución del programa principal seteamos el cerrojo // como abierto a 0 y lanzamos la ejecución de los N hilos // concurrentes </pre>
---	---

El uso de **test and set** implica **espera activa** (busy waiting) o **espera cíclica** (spin waiting).

Esto quiere decir que el proceso no puede hacer otra cosa que no sea esperar hasta que el cerrojo se abra y le permita el paso a su sección crítica.

## Instrucción Exchange

La instrucción intercambia recibe dos parámetros, registro y memoria. Apoyándose en una variable auxiliar, intercambia los valores.

<pre> /* programa exclusión mutua */ const int n = /* número de procesos */; int cerrojo; void P(int i) {     int llavei = 1;     while (true)     {         do exchange (llavei, cerrojo)         while (llavei != 0);         /* sección crítica */;         exchange (llavei, cerrojo);         /* resto */     } } void main() {     cerrojo = 0;     paralelos (P(1), P(2), . . . , P(n)); } </pre>	<pre> //Se declara un número n de procesos //Se declara una variable entera compartida llamada cerrojo  //Para cada ejecución de proceso //Declaramos una variable llave con valor a 1  //Intercambiamos el valor de llave y cerrojo //Mientras que la llave no haya sido utilizada para bloquear el //cerrojo poniendolo a 1... (lo cual quiere decir que no se ha //podido ejecutar la instrucción atómica exchange) //Espera activa  //Se ejecuta la sección crítica una vez se cumple el exchange  //Al salir de sección crítica se ejecuta exchange de nuevo para //dejar el cerrojo abierto de nuevo  //En el programa principal se inicializa el cerrojo a 0 //Y se ejecutan concurrentemente los N procesos </pre>
--	--

Esta instrucción tiene sus ventajas:

- Aplicable a N procesos sobre mono o multiprocesador
- Simple
- Soporta varias secciones críticas usando cada cual su propia variables

Pero tiene serias desventajas:

- Espera activa
- Cabe la posibilidad de inanición debido a que la selección tras la espera activa es arbitraria
- Cabe la posibilidad de interbloqueo

# Semáforos

Dijkstra implementó el algoritmo de dekker y a posteriori lo que se denominan semáforos, que resuelven o mejoran la espera activa dada en los algoritmos de Dekker y Peterson.

Los semáforos se fundamentan en lo siguiente:

- Dos o más procesos pueden cooperar por medio de señales
- Dichas señales pueden obligar a parar un proceso en un lugar específico hasta que reciba una señal específica
- Cualquier requisito de coordinación puede satisfacerse con una estructura de señales
- Para dicha señalización se usan variables especiales llamadas semáforos

## ***Semáforo General***

Para transmitir una señal usando un semáforo S usamos semSignal(s)

Para recibir una señal el semáforo S usamos semWait(s)

Si la señal no ha llegado aun, el proceso se suspenderá hasta que dicha transmisión se efectúe

Así podemos definir el semáforo como variable con valor entero sobre la que se definen 3 operaciones:

- **Inicialización** a valor **no negativo**
- **semWait(semáforo)** **decrementa el valor del semáforo**. Si el semáforo es **negativo** el proceso en ejecución **se bloquea**, en otro caso continúa su ejecución
- **semSignal(semáforo)** **incrementa el valor del semáforo**. Si el valor es **menor o igual a cero se desbloqueará** uno de los procesos bloqueados por semWait(semáforo)

Dichas operaciones se realizan de modo **atómico**, y el encargado es el propio núcleo del sistema.

Con estas señales se **garantiza** que ningún proceso accederá al semáforo hasta que la operación se haya completado, por tanto se **desactivan las interrupciones**.

## ***Semáforo Binario***

Una versión más simple del semáforo general es el semáforo binario o mutex. Este solo puede tomar 2 valores, abierto o cerrado (1 o 0). Se pueden definir estas 3 operaciones:

- **Inicialización a 0 o 1**
- **SemWaitB() comprueba el valor de mutex** y si es 0 el proceso se bloquea. Si es 1 se cambia a 0 y se sigue ejecutando.
- **SemSignalB() comprueba si hay uno o varios procesos bloqueados** por el semáforo y si los hubiese desbloquea uno de ellos. Una vez no queda ningún proceso bloqueado el valor del semáforo cambia a 1.

A la hora de implementar un mutex binario o un semáforo general no habrá distinción entre uno y otro. Para ambos se usa una cola que mantiene los procesos en espera. Respecto la cuestión de extraer dichos procesos de la cola surge la política FIFO (first in first out) así los procesos van liberándose en el orden en el que fueron entrando (cola).

Un **semáforo** que **incluye** una **política de extracción** de procesos se denomina **semáforo fuerte**.

Uno que **no especifique** el orden de **extracción** se denomina **semáforo débil**.

# Monitores

Los semáforos son una herramienta potente para sincronizar, coordinar y garantizar exclusión mutua, pero también son difíciles de interpretar en la lectura de un código de modo global.

Para facilitar este aspecto **Hansen** y **Hoare** propusieron una primitiva de sincronización de mayor nivel, conocida como monitor.

Un monitor es una **colección de procedimientos, variables y estructuras de datos (similar a una clase en orientación a objetos)** agrupados en un módulo o paquete con estas características:

- Las **variables locales solo** son **accesibles** por los **métodos del monitor** (encapsulación private)
- Un **proceso entra** en el **monitor invocando** uno de sus **métodos**
- **Solo un proceso puede ejecutar en el monitor al mismo tiempo**, por tanto, cualquier otro proceso que requiera manejar el monitor queda bloqueado hasta que esté disponible.

Las dos primeras características se asemejan al software orientado a objetos.

Al cumplir la premisa de un solo proceso al mismo tiempo, el monitor proporciona exclusión mutua.

Los monitores son construcción de un lenguaje, por lo que el compilador sabe que son especiales y pueden manejar llamadas a procedimientos del monitor de modo distinto a las llamadas a otros procedimientos. Por lo general, en una llamada a método de monitor lo primero que se comprueba es si hay otra instancia de este en ejecución. Si es así la actual llamada al método se postergará hasta que la instancia actual haya terminado.

En resumidas cuentas, el monitor contendrá tantos métodos como secciones críticas y nunca dos procesos ejecutarán sus secciones críticas a la par.

## Interbloqueo e inanición

El **interbloqueo** consiste en un **bloqueo permanente** de un conjunto de **procesos** que **compiten por un recurso o se comunican entre sí**. Se da cuando, por ejemplo un par de procesos requieren de la liberación de un recurso por parte de otro para continuar, y en estado de espera quedan eternamente. Las condiciones para que se de son:

- **Exclusión mutua:** Solo un proceso usa el recurso a la par.
- **Retención y espera:** Un proceso mantiene reservados los recursos mientras aguarda otros.
- **Sin expropiación:** No cabe posibilidad de expropiar de un recurso a un proceso que lo posea
- **Espera circular:** Existe una lista cerrada de procesos en la que cada cual posee un recurso necesitado por otro proceso.

La **inanición** es similar al interbloqueo dado que algún proceso no llegará jamás a usar la sección crítica. Esto se puede dar porque **un proceso “secuestre” el uso de procesador** para el solo y **jamás** salga para **dar oportunidad a otro proceso**. Para evitar este problema se utilizan algoritmos de planificación, así los procesos con más prioridad no ocuparán permanentemente el uso del procesador y dependiendo de la política de planificación, dejará paso a unos u otros procesos.

# Categorías de recursos

Los recursos pueden ser consumibles o reutilizables:

- **Consumibles:** Son aquellos que **pueden producirse y consumirse**. No hay límite en el número de recursos consumibles de ningún tipo. Suelen ser creados por un proceso productor y consumido por un proceso consumidor que tras adquirirlo elimina el recurso. Ejemplos de este tipo son las interrupciones, señales, mensajes y **buffers de E/S**.
- **Reutilizables:** Son aquellos que solo se **pueden usar** de forma segura solo **un proceso a la vez**, y que **no se destruye tras su uso**. Ejemplos de este tipo son canales de E/S, memoria principal y secundaria, **ficheros...**

## Paso de mensajes

Cuando los procesos interaccionan entre sí se deben sincronizar y comunicar. Para cumplir con ambos objetivos se usa el paso de mensajes, que se puede usar tanto en sistemas multiprocesador con memoria compartida como monoprocesador.

Las funciones de paso de mensaje incluyen el par de primitivas:

- Send(destino,mensaje)
- Receive(origen,mensaje)

### **Sincronización**

La comunicación de mensajes entre procesos requiere de sincronización, especificando que sucede con un proceso tras ejecutar una primitiva send o receive. Se suelen dar las siguientes combinaciones:

- **Envío bloqueante, recepción bloqueante:** Al ejecutar send, el proceso origen espera e igual sucede con el proceso receptor cuando ejecuta receive. Conocido como rendezvous.
- **Envío no bloqueante, recepción bloqueante:** Al ejecutar send el proceso origen continúa con su ejecución, el proceso receptor sin embargo una vez ejecuta receive queda bloqueado hasta recibir el mensaje.
- **Envío no bloqueante, recepción no bloqueante:** Tanto el proceso origen como el de destino ejecutan sus primitivas y no se bloquean.

**Lo más natural en concurrencia es el send no bloqueante**, por **ejemplo** un proceso que ejecuta una **petición de impresión** y continúa con su ejecución

Un peligro del send no bloqueante es que un error provoca que los procesos generan mensajes repetidamente y consumen recursos del sistema.

**La primitiva receive bloqueante suele ser la más natural en concurrencia.** Generalmente un proceso que quiere un mensaje ha de esperar a dicha información antes de continuar.

Si un mensaje se pierde o el proceso origen falla, el proceso receptor puede esperar eternamente, esto se puede resolver con un receive no bloqueante, con el peligro de que si un mensaje es enviado después de que un proceso haya realizado su receive, este mensaje se puede perder.

### **Direccionamiento**

**En el envío** es necesario **especificar** que **procesos** han de **recibir el mensaje**, al igual, en la mayoría de implementaciones se permite al **proceso receptor indicar el origen**



del mensaje.

Existen distintos esquemas para especificar procesos en send y receive:

➤ **Direccionamiento directo:**

- La primitiva send **identifica** el proceso **destinatario**.
- La primitiva receive puede indicar **explícitamente el emisor**.
- La primitiva receive puede indicar **implícitamente el emisor** una vez se **concluya la recepción**, por ejemplo, en un servidor de impresión que no se sabe que proceso será el que le haga dicha petición, se reconocerá al emisor cuando se concluya la recepción.

➤ **Direccionamiento indirecto:**

- Los mensajes no se envían directamente entre los procesos, sino que existen unos **buzones** o "mailboxes" que actúan de **intermediarios**.
- Dichos buzones son **colas** que **contienen temporalmente** esos **mensajes**
- El desacople de emisor-receptor mediante buzones permite la **flexibilidad** de crear **relaciones**:
  - \* **Uno a uno**: Comunicación privada entre dos procesos
  - \* **Muchos a uno**: Útil para interacciones cliente/servidor, cuando un proceso proporciona servicio a muchos otros
  - \* **Uno a muchos**: Permite difusión de mensajes
  - \* **Muchos a muchos**: Permite a múltiples servidores dar servicio a múltiples clientes.

## **Formato**

El **formato** del mensaje **depende de los objetivos** de la facilidad de mensajería y de si se usa en **computador único** o en **sistema distribuido**.

En **algunos** sistemas se opta por **mensajes cortos de longitud fija** para minimizar la sobrecarga de procesamiento y almacenamiento.

Si se transfieren **grandes volúmenes**, se pueden disponer los **datos en un archivo** y **pasar** como mensaje un **puntero** a este.

Otra solución son mensajes de **longitud variable**.

Los **campos** que pueden contener un mensaje son:

➤ **Cabecera**

- Tipo de mensaje

- Destino
  - Origen
  - Longitud de mensaje
  - Información de control (punteros por ejemplo)
  - Prioridad
- **Cuerpo:** Contenido real del mensaje

### ***Disciplina de Cola***

La disciplina mas **sencilla es FIFO**, aunque puede no ser suficiente si algunos mensajes son más urgentes que otros.

Una alternativa es **especificar prioridad** de mensaje **según** el **tipo o** según el **emisor**.

Otra alternativa es **permitir** al **receptor inspeccionar la cola** y seleccionar que mensaje quiere recibir a continuación.

# TEMA 4

## PLANIFICACIÓN

---

### Objetivos

- **Justicia:** Planificar de modo que **cada proceso** tenga su **quanto** de tiempo **en CPU**
- **Productividad:** Acabar el **máximo** posible de **trabajos** por **unidad de tiempo**
- **Tiempo de respuesta** aceptable: El **usuario** ha de **sentir velocidad** en las ejecuciones.
- **Predecible:** La planificación debe obtener **resultados métricos parecidos**, sea cual sea la carga.
- **Coste extra:** El **tiempo** usado en **planificar** ha de ser el **mínimo** posible.
- **Prioridades:** Se deben establecer **prioridades** en **función** de la **importancia** de cada **trabajo**.
- **Ocupación de recursos:** Se debe **maximizar** el **uso** de **recursos** del sistema, dando prioridad a procesos que retienen los recursos, por ejemplo.
- **Aplazamiento indefinido:** Se debe **evitar** la **inanición**.
- **Degradación aceptable:** El algoritmo planificador debe procurar una **degradación paulatina** del sistema, **sin** que llegue a ocurrir un **colapso global**.

# Criterios para la planificación

Los administradores de centros de cómputo se basan en unas **métricas** para verificar el **desempeño de los sistemas**, las cuales sirven para comparar el **rendimiento** de los diversos **algoritmos de planificación**:

- **Rendimiento** o Productividad (throughput): Número de **trabajos por unidad de tiempo**. Depende de la longitud de los procesos.
- **Tiempo de retorno (Tr) o estancia** (turnaround time): Es el tiempo **promedio desde** que se **envía** un trabajo **hasta** que **concluye**. Se incluyen tiempos de ejecución efectiva, espera, E/S...
- **Tiempo de espera (Te)**: Es el **tiempo** que el proceso está **esperando en colas** hasta que se le asigna su cuanto de CPU. ( $Tr - \text{tiempo en cpu} - \text{tiempo E/S}$ )
- **Tiempo de servicio (Ts)**: Es cuanto de tiempo requerido de CPU, o el tiempo que se consumiría si fuese el único proceso existente. ( $Tr - Te$ )
- **Tiempo de respuesta**: Es el tiempo **transcurrido desde** que se **envía** un proceso **hasta que se termina** de ejecutar la **primera instrucción** (se recibe el primer resultado). Es una buena medida ya que el usuario quiere ver resultados cuanto antes.
- **Uso de CPU o eficacia: Porcentaje** de ocupación de **CPU**. Cuanto menor tiempo esté la CPU ociosa más eficacia.

# Modos de decisión

Los **algoritmos de planificación se dividen en 2 categorías** respecto como manejan las interrupciones del reloj (instantes en los que se selecciona un proceso)

## *Sin expulsión o apropiativos*

El **planificador selecciona un proceso** para ejecutar y deja que se ejecute. Este **se ejecutará hasta su conclusión**, incluso habiendo interrupciones durante su ejecución, no se tomarán decisiones de planificación.

## *Con expulsión o apropiativos*

El **planificador selecciona un proceso** para ejecutar y la decisión de **expulsión** de proceso se toma **según**:

- Si el proceso ha **consumido su cuanto de CPU**
- Si llega un **nuevo proceso con más prioridad**
- Si llega una **interrupción** que pasa un proceso de **bloqueado a listo**

Si sigue en ejecución hasta el final del cuanto, se suspende y el planificador elige otro proceso para ejecutar si hay alguno disponible.

Las **políticas expulsivas** tienen **más sobrecargas que las no expulsivas (cambios de contexto)** pero proporcionan mejor servicio al cómputo total de procesos, ya que evitan el “secuestro” de la CPU por parte de uno y esto provoque inanición.

El coste de expulsión suele ser bajo usando mecanismos eficientes, con toda la ayuda hardware posible, y con gran espacio en memoria principal para poder almacenar procesos y hacer cambios de contexto rápidamente.

# Tipos de planificación

El **objetivo** de la planificación es **asignar procesos a ejecutar** por el procesador **a lo largo del tiempo**, de modo que se cumplan los criterios de planificación (tiempo de respuesta, rendimiento, eficiencia).

En muchos sistemas se divide en las funciones a largo, medio y corto plazo.

## *Planificación a largo plazo*

**Determina** que **programas se admiten** en el sistema para su procesamiento. Así se **controla** el **grado de multiprogramación**.

Una vez admitido un **programa se convierte en proceso** y se añade **a la cola de planificación a corto plazo**. En algunos sistemas un proceso recién creado comienza en la zona de intercambio, añadiéndose así a la cola de planificación a medio plazo.

La decisión de cuando crear un proceso se toma dependiendo del grado de multiprogramación deseable. Cuanto **más procesos creados, menor será el porcentaje** de tiempo asignado **de CPU**.

De este modo, el planificador a largo plazo limita el grado de multiprogramación a fin de dar servicio satisfactorio al conjunto de procesos. Al acabar un trabajo, el planificador decide si añadir nuevos trabajos.

La decisión de que trabajo será el próximo se puede basar en una cola FIFO o en una herramienta de gestión de rendimiento. El criterio puede incluir la prioridad, tiempo estimado de ejecución, requisitos de E/S...

## *Planificación a medio plazo*

**Es parte de la función de intercambio**. Frecuentemente la **decisión de intercambio** se **basa** en la **necesidad de gestionar el grado de multiprogramación**.

En un sistema sin memoria virtual, la gestión de memoria es otro aspecto a tener en cuenta.

Así, la decisión de almacenar un proceso en memoria, contará con las necesidades de la memoria de los procesos que están fuera de ella.

## *Planificación a corto plazo*

El planificador a largo plazo ejecuta relativamente con poca frecuencia con toma de decisión de grano grueso a la hora de admitir o no nuevos procesos.

El planificador a medio plazo se ejecuta con más frecuencia para planificar intercambios.

El planificador a **corto plazo, conocido como activador**, se ejecuta mucho más frecuentemente, **tomando decisiones** de grano fino sobre **que proceso** será el **próximo a ejecutar**.

Siempre **se invoca** cuando ocurre un **evento** que pueda **bloquear el proceso actual** y proporcione oportunidad de expulsión de el proceso actual en favor de otro. Dichos eventos pueden ser:

- **Interrupciones de reloj**
- **Interrupciones de E/S**
- **Llamadas al sistema**
- **Señales (semáforos)**

# Algoritmos de planificación

En muchos sistemas, cada **proceso tiene** asignada su **prioridad** y **en función de esta** el **planificador escogerá** entre procesos.

Cuando se va a **seleccionar en la planificación**, el planificador **comienza** con la **cola de procesos listos de mayor prioridad** Cola Listos 0 (**CL0**). Si hay varios procesos se selecciona uno **según una política de planificación**.

Si CL0 está vacía, se examinará CL1 y así sucesivamente.

Los sistemas de **planificación con prioridades** tienen el **problema** de que los procesos de **baja prioridad pueden sufrir de inanición** (sucede cuando siempre hay procesos con mayor prioridad listos para ejecutar).

## ***FCFS (First Come First Served)***

El primero en llegar, en primero en servirse, **también conocida como FIFO**. Es un sistema de colas estricto.

Cuando el **proceso** que se ejecuta **acaba su cuanto** de CPU, **se toma el proceso que ha estado más tiempo en la cola de listos**.

Funciona mucho **mejor para procesos largos que para procesos cortos**.

Esta política **no hace un uso eficiente de CPU ni de E/S**, debido a que mientras que un proceso limitado por procesador se ejecuta, la E/S no puede ser usada por los procesos bloqueados, ya que deberán esperar a que el proceso en ejecución acabe su tarea.

Del mismo modo si un proceso en ejecución queda bloqueado, el procesador quedará ocioso.

FCFS **no es una alternativa atractiva** para sistemas monoprocesador, aunque **combinado con esquemas de prioridad consigue una planificación eficaz**.

## ***Round Robin (Turno Rotatorio)***

Esta política **se basa en el cuanto de tiempo** para asignar un **turno cíclico de CPU a cada proceso**.

Cada vez que el intervalo de tiempo determinado (**quanto**) **se ha cumplido**, se produce una **interrupción de reloj** que **intercambia el proceso en ejecución por el siguiente** en la cola de listos.

Es conocida también como **cortar el tiempo (time slicing)** ya que a cada proceso se le



da una “rodaja de tiempo” antes de ser expulsado.

El **factor clave** en esta política es el tamaño del **quanto**. Si es **muy pequeño** el proceso se moverá rápido, pero los cambios de contexto, señal de interrupción de reloj y demás funciones de planificación **pueden sobrecargar el sistema**.

Por este motivo, lo ideal es que el **quanto de tiempo** sea **ligeramente mayor** que el **tiempo** requerido **para una función típica del proceso**, así conseguiremos en una única unidad de tiempo abarcar toda la tarea.

Un **quanto más largo** que el **proceso más largo** en ejecución **degenera** el Round Robin **en FCFS**.

El round robin es efectivo en sistemas de tiempo compartido de proposito general o sistemas de procesamiento transaccional.

Una desventaja de este sistema es que trata desigualmente procesos limitados por procesador y limitados por E/S en detrimento de estos últimos que tienen menos unidades de tiempo asignadas en procesador.

### ***SPN (Shortest Process Next)***

**Primero el proceso más corto**, es una política **no expulsiva** que **elige el proceso con tiempo de procesamiento más corto**. Así, este se situará en la **cabeza de la cola** por delante de los más largos.

El **rendimiento global mejora** significativamente en **tiempo de respuesta**, sin embargo, el **tiempo de respuesta varía impredeciblemente**.

Esta política presenta el problema de la **necesidad** de conocer o **estimar el tiempo de procesamiento** requerido por cada proceso. En trabajo por lotes el sistema puede requerir que el **programador estime ese valor** y lo proporcione.

Si la **estimación** es mucho **menor** que el tiempo actual de ejecución, el **sistema** podría **abortar el trabajo**.

En **entorno de producción**, donde se ejecutan frecuentemente los mismos trabajos, se pueden **recoger estadísticas**, pudiéndose calcular una **media del tiempo de ejecución** por cada ráfaga.

CONTINUA...

### ***SRT (Shortest Remaining Time)***

Es una **versión expulsiva del SPN**, en cuyo caso el **planificador** siempre **escoge el proceso con menor tiempo** de proceso **restante** esperado.

**Al unirse un nuevo proceso** a la cola de listos, puede tener un **tiempo restante menor que el actual** en ejecución, por tanto, el **planificador expulsa al actual** para dar paso a este.

Al igual que en SPN, el **planificador debe tener la estimación de tiempo de proceso** para realizar la función seleccionada, y cabe la **posibilidad de inanición en procesos largos**.

SRT **no favorece a procesos largos**, tal y como sucede en FCFS. A diferencia de Round Robin, **no hay interrupciones adicionales, reduciéndose sobrecarga**, aunque hay que **almacenar tiempos de servicio** transcurridos, lo cual **si produce sobrecarga**.

**SRT debería mejorar SPN** porque da **preferencia a trabajos cortos sobre los largos**.

## ***Retroalimentación***

**Si no es posible conocer el tiempo** de servicio de varios procesos, no es posible utilizar SPN ni SRT.

Un modo de establecer preferencia en trabajos cortos es **penalizar a los trabajos** que se **ejecutan durante más tiempo**, basándonos en un sistema de **prioridades dinámico**.

Cada vez que un **proceso** es **expulsado de CPU**, se va **añadiendo** a una **cola de menor prioridad**, de modo que un proceso corto no llegará a bajar mucho en el árbol de prioridades.

En cambio, un **proceso largo**, irá **degradando su prioridad** gradualmente, favoreciéndose así a los procesos cortos sobre los viejos y largos. **Dentro de cada nivel de cola se emplea FCFS**.

Una vez un **proceso** entra en la **cola de menor prioridad**, no podrá descender más, por lo que **entrará continuamente a la misma cola** hasta que complete su trabajo.

Este sistema es conocido como **retroalimentación multinivel**.

Existe un problema en este esquema y es que la estancia para **procesos largos puede eternizarse**, incluso puede llegar a existir inanición si no paran de llegar trabajos nuevos al sistema.

**Para solucionarlo** se han de asignar **distinto quanto** de tiempo **en cada cola**, para ser más exactos, en la **Cola<sup>1</sup> se darán 2<sup>1</sup> quantos** de tiempo de procesador.

## ***FSS (Faire Share Scheduling)***

Todos los algoritmos anteriores tratan los procesos listos como un conjunto de procesos entre los cuales se elige el próximo a ejecutar. Este conjunto se puede romper con el uso

de prioridades pero es homogéneo.

En un SO multiusuario, si las aplicaciones o trabajos se organizan como múltiples procesos o hilos, existe una estructura en la colección de procesos que no se contemplan en planificadores tradicionales.

Desde la visión del usuario, la preocupación no es como ejecutar un simple proceso, si no su conjunto de procesos que conforman la aplicación. Así se pueden tomar decisiones de planificación basados en dichos conjuntos.

Este enfoque se conoce como planificación de contribución justa.

Este concepto se puede extender a un grupo de usuarios, incluso si cada usuario se representa con un solo proceso.

CONTINUA...

# Comparación de rendimiento

## Planificación Unix Tradicional

Este algoritmo de planificación se diseñó para proporcionar buenos tiempos de respuesta a usuarios interactivos, a la par que asegura que los trabajos de fondo de baja prioridad no sufran inanición.

Aunque se reemplazó en los sistemas Unix modernos, es bastante representativo de los algoritmos prácticos de planificación de tiempo compartido.

Esta planificación emplea reatrolimentación multinivel y planificación Round Robin para cada cola de prioridad. El cuanto de tiempo para expulsión es de 1 segundo. La prioridad se basa en el tipo de proceso e histórico de ejecución.

## Planificación multiprocesador

Cuando un **computador** tiene **más de un procesador**, la planificación debe **distinguir entre nivel de proceso y a nivel de hilo**.

Los sistemas multiprocesador se pueden clasificar en:

- **Débilmente acoplados** o distribuidos: Colección de **sistemas** relativamente **autónomos**, cada cual con su memoria y canales de E/S.
- De **funcionalidad especializada**: Por ejemplo, un procesador de E/S. Hay un **procesador maestro y procesadores especializados controlados por este** y al que proporcionan servicio.
- **Fuertemente acoplado**: Son un **conjunto de procesadores** que comparten memoria y **están bajo el control de un único SO**. Este es el que nos concierne, específicamente en sus aspectos de planificación.

## Granularidad

Un método **para distinguir multiprocesadores de otras arquitecturas** es **considerando la granularidad** o frecuencia de sincronización entre procesos. Se pueden distinguir 5 categorías.

### *Paralelismo independiente*

**No existe sincronización explícita** entre procesos, ya que **cada procesador** representa un **trabajo independiente**.

Se da en en **sistemas de tiempo compartido**, donde cada usuario desarrolla sus trabajos con una aplicación y el **multiprocesador** proporciona el **mismo servicio que un monoprocesador multiprogramado**.

Dado que hay más de un procesador, el tiempo de respuesta medio de los usuarios sera menor.

### ***Paralelismo grano grueso y muy grueso***

Hay **sincronización** pero a un nivel muy burdo.

Este tipo de situación **trata** sencillamente como un **conjunto de procesos concurrentes** ejecutando en un **monoprocesador multiprogramado**, y **se puede dar en un multiprocesador sin** apenas **cambios en el software**.

Generalmente, una colección de procesos concurrentes que necesitan sincronizarse pueden beneficiarse de la arquitectura multiprocesador.

En caso de que la **comunicación** entre procesos sea **poco frecuente**, un **sistema distribuido puede dar un buen soporte**, aunque si la **comunicación** es **algo más frecuente** se **puede** producir una **sobrecarga de comunicaciones** y puede **mermar** la mejora de **velocidad**.

En el último caso, una **arquitectura multiprocesador** proporciona un soporte **más eficaz**.

### ***Paralelismo grano medio***

Los **paralelismos anteriores pueden proporcionarse** en un monoprocesador multiprogramado o en multiprocesador **con apenas modificaciones en la planificación, pero cuando** se trata de **planificación de hilos** es **necesario un replanteo**.

Varios **hilos de una aplicación interactúan** muy **frecuentemente**, por tanto las planificación respecto a hilos **influyen directamente en las prestaciones** de la aplicación.

### ***Paralelismo grano fino***

Representa un **uso mucho más complejo del paralelismo que el existente en el uso de hilos**. Esta es a día de hoy un **área especializada y fragmentada** con muchas propuestas distintas.

# Aspecto de diseño

En multiprocesador la planificación involucra los siguientes aspectos:

## *Asignación de procesos a procesadores*

Partiendo de que la **arquitectura multiprocesador es uniforme**, debido a **cada procesador** tiene la **misma ventaja** a la hora de **acceder a memoria principal o a E/S**, el enfoque más simple de planificación consiste en **tratar los procesos como recursos colectivos**, y **asignarlos a los procesadores** según demanda.

Surge entonces la cuestión de si deben asignarse **estática o dinámicamente**.

Si se hace **estáticamente**, un proceso **queda asignado a un procesador hasta su final**, manteniéndose en una cola a corto plazo dedicada a ese procesador.

La **ventaja** de esta estrategia es que se **sobrecarga menos la función de planificación**, la **desventaja** es que se permite a los otros **procesadores** estar **ociosos**.

Para evitar dicha situación se **puede usar una cola común** y luego ser **procesados** por **cualquier procesador disponible**.

De este modo un proceso durante su vida ha podido ser ejecutado por distintos procesadores en distintos momentos.

En **arquitecturas** de memoria **fuertemente acopladas**, la información de **contexto de todos los procesos está disponible para todos los procesadores**, lo que conlleva a que el coste de planificación de un proceso es independiente a qué procesador es asignado.

**Otra opción es el balance dinámico de carga**, donde los hilos se mueven de la cola de un procesador a la de otro, enfoque usado en Linux.

**Para** realizar las **asignaciones de procesos a procesadores** se puede **usar** el enfoque **maestro/esclavo o camaradas**.

**Maestro/Esclavo** ejecuta en **un procesador** concreto ciertas **funciones del sistema** y los **otros procesadores** ejecutan **programas de usuario**. El maestro planifica trabajos. Una vez el proceso está activo, si un esclavo requiere de un servicio, como una E/S, solicita al maestro y espera a que el servicio se realice.

Sus desventajas son que si **falla el maestro** se **cae el sistema** y que la dependencia del maestro puede provocar un **cuello de botella** en el rendimiento del sistema.

En la arquitectura en **camaradas** el **núcleo del SO** puede ser **ejecutado** en **cualquier procesador**, que a su vez **se autoplanifica** desde la colección de procesos disponibles.

**Este enfoque complica el SO**, ya que debe asegurar que 2 procesadores no seleccionen el mismo proceso y que estos no se extravíen. **Se han de usar técnicas de sincronización** en demanda de recursos.

Entre estas dos técnicas hay una **gama de opciones**, como tener un grupo de procesadores maestros.

### ***Uso de multiprogramación en monoprocesadores***

Los **multiprocesadores tradicionales** realizan **sincronizaciones de grano grueso o independientes**, por lo que **cada procesador individual debe poder cambiar entre procesos** para conseguir **alta utilización y mejor rendimiento**.

Sin embargo, **en aplicaciones de grano medio** en **multiprocesador con muchos procesadores** el **objetivo deja de ser** el de conseguir una **ocupación de todos los procesadores** durante todo el tiempo que sea posible, lo que **se busca** es **proporcionar el mejor rendimiento medio de las aplicaciones**.

Una **aplicación con muchos hilos** ejecutará con **dificultad**, a menos que sus hilos estén **pensados para ejecución concurrente**.

### ***Activación de procesos***

En la **planificación monoprocesador multiprogramado** se usan **prioridades o algoritmos basados en el uso pasado** para mejorar el rendimiento.

En los **sistemas multiprocesador** la **algorítmica** de elección puede ser **innecesaria e incluso contraproducente**. Con un **enfoque más simple** puede conseguirse **eficacia y menos sobrecarga**.

En **planificación de hilos** entran en juego **nuevos aspectos** más importantes que prioridades o historiales de ejecución.



# **Planificación de hilos**

*Compartición de carga*

*Planificación en pandilla*

*Asignación de procesador dedicado*

*Planificación dinámica*