Universitat de les Illes Balears

Escuela Politécnica Superior

Grado en Ingeniería Informática

Sistemas Operativos I

Apuntes

Luis Barca Pons

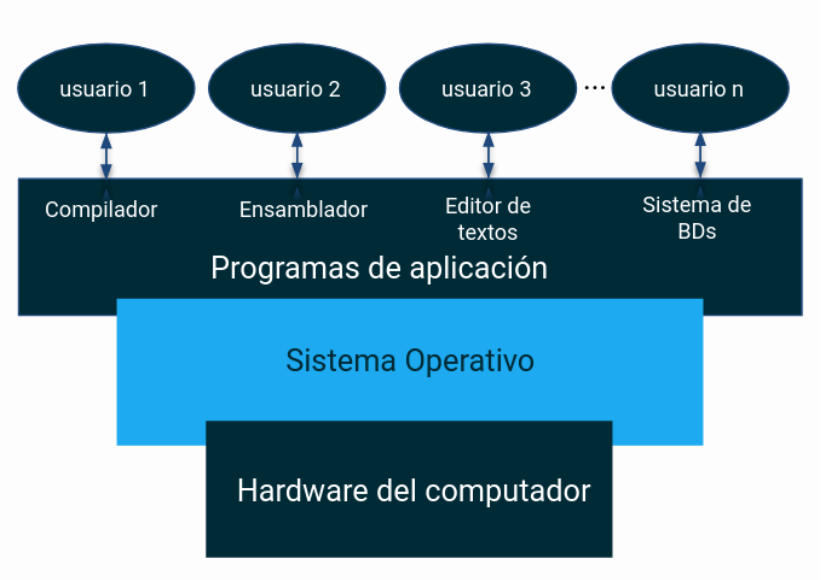
Curso 2021/22

Indice

Tema 1: Aspectos generales de los SSOO

1.1. Objetivos, funcionalidad y aspectos de diseño de los SSOO

Un sistema operativo es un **programa o conjunto de ellos** que controlan la ejecución de aplicaciones y programas. Actúa como **interfaz** entre las aplicaciones (el software) y los componentes (el hardware) del computador.



1.1.1. Funciones y objetivos

- **Facilidad de uso**
  - El usuario ve el sistema operativo como un conjunto de aplicaciones
  - El sistema operativo oculta detalles de hardware al programador y le proporciona una interfaz apropiada para utilizar el sistema.
  - Proporciona servicio a las siguientes áreas:
    - Desarrollo de programas
    - Acceso a dispositivos de E/S
    - Acceso al sistema
    - Contabilidad
    - Ejecución de programas
    - Acceso controlad a ficheros
    - Detección y respuesta a errores
- **Eficiencia**

  Es necesario hacer una correcta gestión de los recursos utilizados para garantizar una eficiencia en las siguientes acciones:
  - Transporte
  - Almacenamiento y procesamiento de datos
  - Control de estas funciones
- Capacidad de **evolución**

  Un sistema operativo ha de evolucionar debido a actualizaciones de hardware y en respuesta a los nuevos servicios que demandan los usuarios o necesidades de los gestores de sistema. Todo ello para resolver fallos que puedan surgir o requisitos de diseño nuevos a implementar, así como la modularidad, interfaces entre módulos y documentación.

1.2. Conceptos básicos

- Procesos

  Un proceso es un programa en ejecución donde **cada proceso dispone de su propio espacio de direcciones**. Un subproceso o **hilo** es una tarea o ramificación de un proceso. **Los hilos de un proceso comparten datos y espacio de direcciones**.

- Multiprogramación / Multiprocesos / Multihilo

  La **multiprogramación** es una técnica por la que dos o más procesos pueden alojarse en memoria principal y ser ejecutados concurrentemente por el procesador. La ejecución de los procesos (o hilos) se va alterando en el tiempo a tal velocidad que da la impresión de realizarse en paralelo, pero realmente el procesador solo puede ejecutar un proceso a la vez. Esto se conoce como pseudoparalelismo. Esto permite el servicio interactivo simultáneo entre varios usuarios de manera eficiente. Aprovecha los tiempos que los procesos pasan esperando a que se completen sus operaciones de E/S y así aumenta la eficiencia en el uso de la CPU.

  Las direcciones de los procesos son relativas. El programador no se preocupa por saber donde estará el proceso, dado que el sistema operativo es el que se encarga de convertir la dirección lógica en física.

  Los **multiprocesos** o multiprocesamiento es la capacidad que tiene el sistema operativo de gestionar diversos procesadores mara mantener la carga de trabajo de cada uno equilibrada; e incluso reasignar de forma dinámica los recursos de memoria y dispositivos para aumentar la eficiencia.

  El **multihilo** o *multithread* es la capacidad del sistema operativo de dar soporte a múltiples hilos de ejecución en un solo proceso.

- Multiprocesador

  Se denomina **multiprocesador** a un ordenador que permite abrir programas en más de una CPU. Este puede ejecutar de manera simultánea varios hilos pertenecientes a un mismo proceso o bien a procesos diferentes.

- Multiusuario

  Se denomina multiusuario a la característica de un sistema operativo o programa que permite proveer servicio y procesamiento a múltiples usuarios simultáneamente. Los usuarios pueden acceder de manera simultánea al mismo ordenador desde diferentes terminales conectados directamente al mismo; por lo que normalmente serán **sistemas de tiempo compartido**.

- Tiempo compartido

  Es el hecho de **compartir de manera concurrente** un recurso computacional entre muchos usuarios por medio de tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del sistema operativo, permitiendo a este reducir el tiempo de respuesta del computador y limitar el uso de la CPU por parte de un proceso dado.

1.3. Estructuras de SSOO

Entendemos por estructura del sistema operativo a los componentes del computador y su forma de interconexión con el propio SO. Podemos encontrar dos estructuras:

- **Monolítica**

  Implementan los cuatro componentes fundamentales (planificación de procesos, administración de memoria principal, gestión de E/S y la administración de ficheros) en el **núcleo** y se gestionan de modo **privilegiado**.

  - Ventajas
    - Alto rendimiento
    - Peticiones entre los componentes se reducen a invocaciones de funciones
  - Desventajas
    - Mucho código ejecutándose en modo privilegiado
    - Un error en el núcleo del SO obliga a reiniciar el sistema por completo.
  - Ejemplos: MS-DOS, UNIX, GNU/Linux

- **Microkernel**

  Implementan en su núcleo únicamente la planificación de procesos, la gestión de interrupciones ()parte básica de la gestión de E/S que necesariamente se tiene que realizar en modo privilegiado) y la comunicación entre procesos. Por tanto, la administración de memoria principal, la gestión de E/S y la gestión de ficheros se ejecuta en modo **usuario**.

  Cuando un proceso cualquiera solicita un servicio a través de una llamada al sistema, el micronúcleo canaliza la petición al proceso servidor correspondiente. Dicha comunicación se efectúa mediante **mensajería**.

  - Ventajas
    - Ejecución de menos código en privilegiado (más fiables)
    - Un error en el código de un proceso servidor no implica tener que reiniciar el sistema por completo.
  - Desventajas
    - Peor rendimiento, ya que cualquier petición requiere de mensajería.
  - Ejemplos:
    - Minix 2 y 3

1.4. Interrupciones

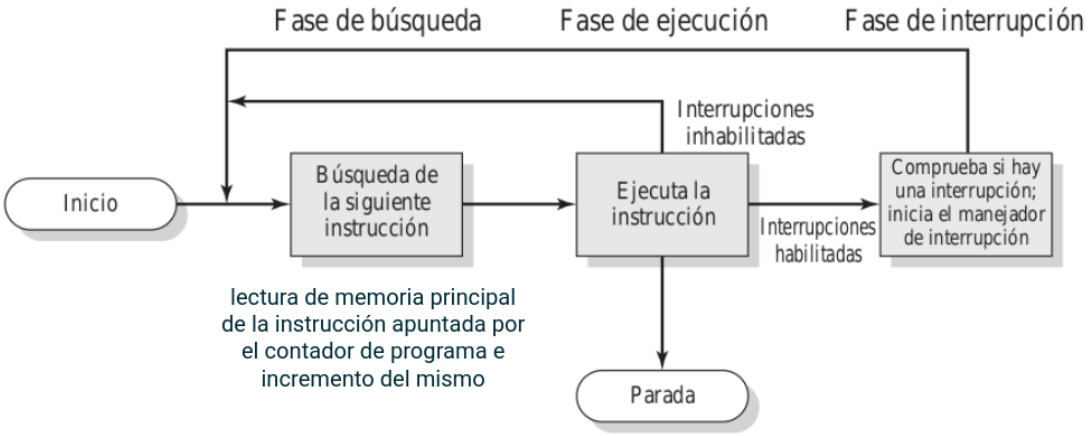
Una interrupción es una **señal de hardware** que llega a la unidad de control para avisar que se ha producido un evento. El sistema operativo toma el control del procesador, suspendiendo o no, la ejecución del proceso activo. Esto permite una mejora en la eficiencia del procesamiento, ya que permite la ejecución de operaciones de E/S y el uso del procesador **simultáneamente**.

Este proceso, en caso de ser suspendido por la CPU, esta podrá reanudarlo en el punto donde fue interrumpido.

1.4.1 Tipos de interrupciones

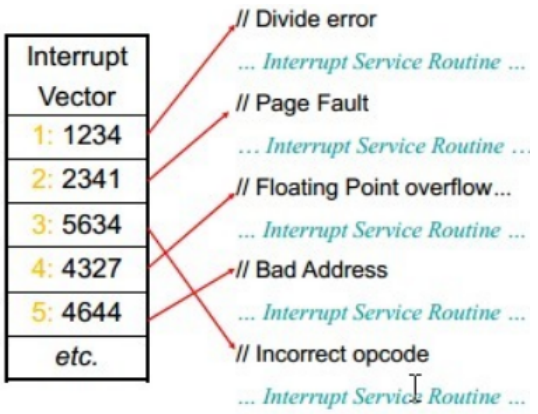
- **Programa**: generadas por la ejecución de alguna instrucción tal como:
  - Desbordamiento aritmético u *Overflow*
  - División por cero
  - Intento de ejecutar alguna instrucción ilegal de la máquina
  - Referencia fuera de espacio de memoria permitido al usuario
- **Reloj**: generadas por el reloj interno del procesador.
- **E/S**: generadas por el controlador de E/S e indican el inicio/fin de una operación de E/S.
- **Fallo de hardware**: generadas por un error de paridad, falta de energía.

1.4.2. Ciclo de instrucción con interrupciones

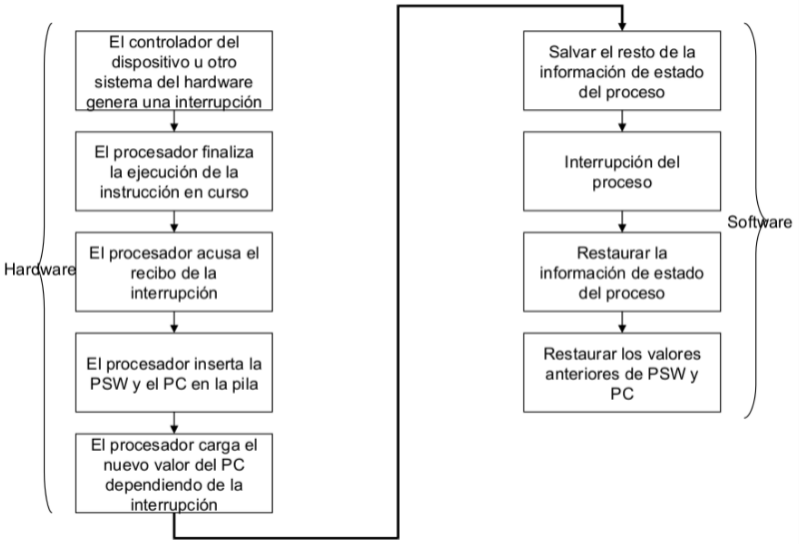


1.4.3. Transferencia de control al ISR

- Cuando se produce una interrupción, el control es transferido a la rutina de servicio de interrupciones (ISR) correspondiente a ese tipo de interrupción.
- Se utiliza la tabla de vectores de interrupciones para determinar qué ISR ejecutar, basándose en el tipo de interrupción, que estará representado por un código.
- Ejemplo de tabla de interrupciones
  - El tipo de error actúa de índice de la tabla
  - Cada entrada contiene la dirección de la ISR correspondiente a ese error

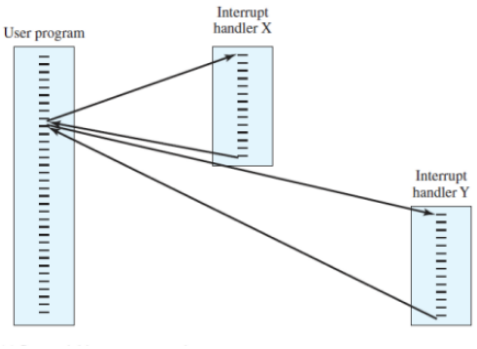


1.4.4. Tratamiento de una interrupción

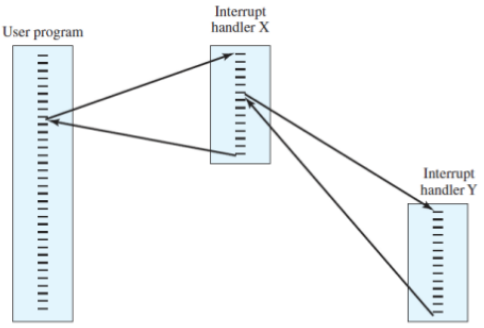


1.4.5. Estrategias de múltiples interrupciones

- Inhabilitarlas
  - El procesador ignorará cualquier nueva interrupción que se produzca
  - Se almacenarán y permanecerán pendientes de ser procesadas, cuando las nuevas interrupciones sean de nuevo habilitadas.
  - Manejo secuencial sin tener en cuenta la prioridad o urgencia de las interrupciones



- Establecer prioridades
  - Permitir que una interrupción de más prioridad e interrumpa la ejecución de un manejador de interrupciones de menor prioridad

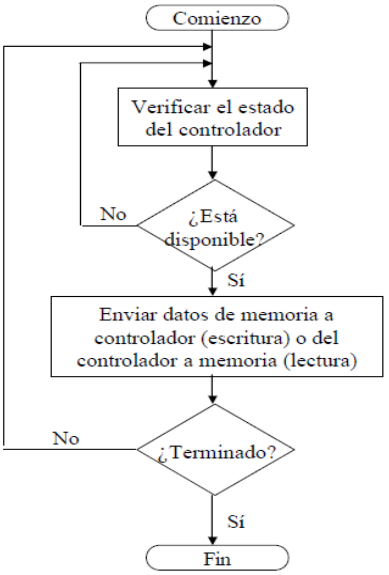


1.5. Entrada/Salida

Podemos diferenciar los tipos de organización del sistema de E/S según la interacción computadora-controlador, de la siguiente manera:

	Sin interrupciones	**Con interrupciones**
Transferencia E/S a memoria a través de la CPU	E/S Programada	E/S dirigida por interrupciones
Transferencia E/S directa a memoria		Acceso directo a memoria (DMA)

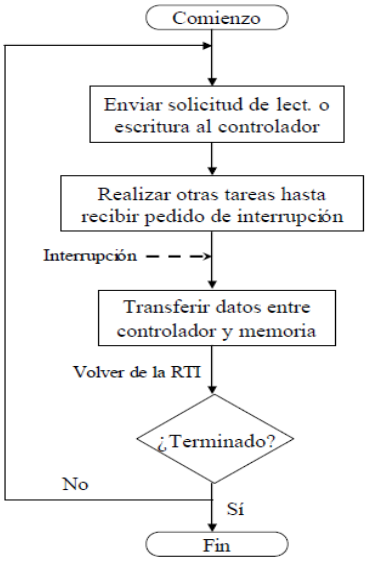
1.5.1. E/S Programada o *Polling*



1. El procesador envía un mandato de E/S a petición de un proceso a un módulo de E/S
2. El proceso realiza una **espera activa** hasta que se complete la operación antes de continuar
3. El módulo de E/S no interrumpe al procesador, sino que este realiza una **consulta periódica** para detectar si el dispositivo está listo

- Desventaja: Consume mucha CPU para dispositivos poco usados
- Aplicación: En desuso excepto para dispositivos que generan muchas interrupciones como tarjetas Ethernet.

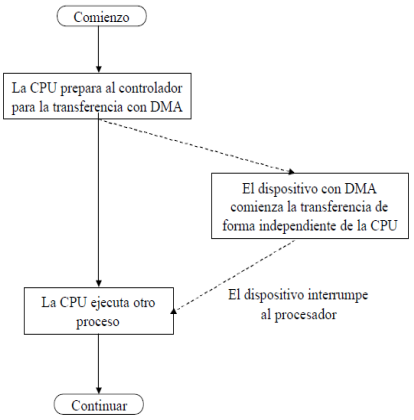
1.5.2. E/S dirigida por interrupciones



1. El procesador emite al controlador un mandato de E/S a petición de un proceso
2. Continúa ejecutando las instrucciones siguientes
3. Es interrumpido por el módulo de E/S cuando este ha completado su trabajo
4. El manejador de interrupciones recibe y maneja ka interrupción
    a. Escoge la rutina a ejecutar
    b. La rutina no tiene acceso a las tablas del espacio de direcciones. Se emplea un buffer (memoria del núcleo) y luego ya se copia a la RAM (doble copia)

- Cada señal de interrupción tiene una determinada prioridad
- Las interrupciones solo son posibles en SSOO apropiativos (pueden arrebatar el uso de CPU a un proceso)

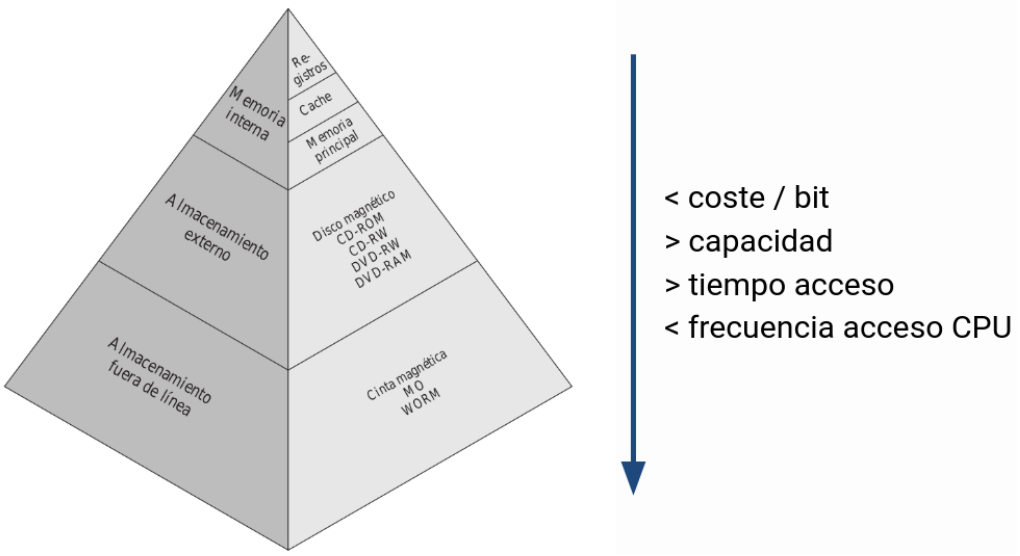
1.5.3. Acceso directo a memoria (DMA)



1. Un módulo de DMA controla el intercambio de datos entre la memoria principal y un módulo de E/S
2. El procesador manda una petición de transferencia de un bloque de datos al módulo de DMA
3. El procesador resulta interrumpido únicamente cuando se haya transferido el bloque completo

- Evita el uso de la CPU transfiriendo los datos directamente entre los dispositivos de E/S y la memoria
- El procesador solo se ve involucrado al principio y al final
- Básico para aprovechar la CPU en un sistema multiprogramado, ya que libera tiempo de la CPU que se puede usar para ejecutar otros programas

1.6. Jerarquía de memoria



1.6.1 Principios de la localidad

- **Temporal**: Si he accedido a una región de memoria, hay una alta probabilidad de que acceda de nuevo a la misma región en poco tiempo.
  - Ejemplo: Bucles, funciones, procedimientos, subrutinas, pilas…
- **Espacial**: Si he accedido hace poco a una región de memoria cercana, hay una alta probabilidad de que acceda a esta.
  - Ejemplos: Arrays, ejecución secuencial…

1.6.2. Memoria caché

Es una **memoria auxiliar** que posee una **gran velocidad y eficiencia**. Normalmente, es utilizada por el microprocesador para reducir el tiempo de acceso a datos, que se utilizan con mayor frecuencia, ubicados en la memoria principal.

El procesador primero verifica la memoria caché. Si no se encuentra en la memoria caché, el bloque de memoria principal conteniendo la información necesaria es movida a la memoria caché.

El procesador primero verifica la memoria cache. Si no se encuentra en la memoria caché, el bloque de memoria principal, que contiene la información necesaria, es movida a la memoria cache para que el procesador pueda manipularla. En cualquier caso podríamos encontrarnos la memoria caché llena, por tanto, usamos **algoritmos de reemplazo** que se encargan de cargar el nuevo bloque.

1.7. Evolución de los SSOO

1.7.1. Sistemas interactivos o procesos en serie

Prehistoria: Los **programadores/operadores** debían interactuar con el hardware del computador sin ayuda externa.

- Accedían directamente a la consola de la computadora
- Se actuaba sobre una serie de micro-interruptores que permitían introducir directamente el programa en la memoria de la computadora (dirección fija)
- Programadores/Operadores
  - Cargan un compilador
  - Un programa fuente
  - Salvan el programa compilado
  - Cargan el ejecutable
- E/S mediante tarjetas perforadas
- Uso por turnos del ordenador
- Se requería mucha planificación y tiempo de configuración

1.7.2. Sistemas de colas simples o procesos por lotes

- **Monitor**: Software que automatiza las funciones del operador. Carga los programas a memoria leyéndolos de una cinta o de tarjetas perforadas, y los ejecuta.
  - La memoria se divide en dos partes:
    - Monitor → Ejecuta en modo sistema o núcleo
    - Programa que se está ejecutando → Modo usuario
  - Características de hardware deseables
    - Protección de memoria
      - La ejecución del programa de usuario no debe alterar el área de memoria del monitor
    - Temporizador
      - Se activa al único de cada trabajo y al expirar se para el programa de usuario y el monitor retoma el control
    - Instrucciones privilegiadas
      - Solo las puede ejecutar el monitor, como las de E/S

        - Interrupciones
            - Flexibilidad para el SO para dejar y retomar el control desde los programas de usuario

- **Procesamiento por lotes**

  En una misma cinta, o conjunto de tarjetas, se cargaban varios programas. Se ejecutaban uno a continuación de otro sin perder apenas tiempo en la transición. Con cada trabajo se incluye un conjunto de instrucciones para el monitor.
    - Lenguajes de control de trabajos (JCL): Conjunto de instrucciones que formaban la cabecera de trabajo y especificaban los recursos a utilizar y las operaciones a realizar por cada trabajo.
        - Ejemplos: FMS (*Fortran Monitor System*), IBYSS y IBM 7094

- **Almacenamiento temporal**

  Simultaneaba la carga del programa o la salida de datos con la ejecución de la siguiente tarea. Para ello se utilizaban dos técnicas:
    - ***Buffering***: Es para **dispositivos que pueden atender peticiones de distintos orígenes**. En este caso los datos no tienen que enviarse completos, pueden enviarse porciones que el **buffer** retiene de forma temporal. También se usan para acoplar velocidades de distintos dispositivos. Así, si un dispositivo lento va a recibir información más rápido de lo que puede atenderla se emplea un **buffer** para retener temporalmente la información hasta que el dispositivo pueda asimilarla. Esto ocurre entre una grabadora de DVD y el disco duro, ya que la primera funciona a una menor velocidad que el segundo.
    - ***Spooling***: Los **datos de salida se almacenan** de manera temporal en una **cola** situada en un **dispositivo de almacenamiento** masivo (spool), hasta que el dispositivo periférico requerido se encuentre libre. De este modo se evita que un programa quede retenido porque el periférico no esté disponible. El sistema operativo dispone de llamadas para añadir y eliminar archivos del **spool**. Se utiliza en **dispositivos que no admiten intercalación**, como ocurre en la impresora, ya que no puede empezar con otro hasta que no ha terminado.
    - **Objetivo**: Disminuir el tiempo de carga de los programas

1.7.3. Sistemas por lotes multiprogramados

La memoria principal albergas múltiples programas de usuario. Al realizar una operación de E/S, los programas ceden la CPU a otro programa. Esto se consigue mediante **algoritmos de planificación**, para que el procesador pueda elegir que programa ejecuta.

1.7.4. Sistema de tiempo compartido

Se comparte el tiempo de procesador entre múltiples usuarios. Cada usuario tiene una terminal online, donde puede enviar comandos y ver su salida de forma interactiva. El sistema operativo entrelaza la ejecución de cada programa de usuario en pequeños intervalos de tiempo o cuantos de computación. Cada usuario tiene la apariencia de que posee la máquina para él solo.

1.7.5. Sistema de propósito general

Son sistemas capaces de operar en lotes, en multiprogramación, en tiempo real, en tiempo compartido y en modo multiprocesador. Tiene un complejo lenguaje de control para los usuarios porque se deben especificar multitud de detalles y opciones. Además, tiene un gran consumo de recursos.

Pero, aparece UNIX. Se reescribe en C y automáticamente es transportable a una amplia gama de computadores.

1.7.6. Sistemas operativos de escritorio

- Difusión de computadores personales
- Objetivo: Aumentar la productividad del usuario
- Aparecen sistemas “amistosos” que permiten la interacción del ratón con menús, iconos y ventanas. No precisan aprender ningún lenguaje de control
    - Ejemplo: System 1 (Macintosh) o Windows 95/98 (sobre MS-DOS)

1.7.7. Sistemas operativos de red y distribuciones

- SSOO Red: Cada computador tiene su propia copia del sistema operativo, los usuarios saben que existen varios computadores, que pueden conectarse explícitamente a diferentes máquinas remotas, para transferir archivos, hacer búsquedas…
- SSOO Distribuidos: Se dispone de varios computadores conectados en red, compartiendo información y periféricos de forma transparente para el usuario

Tema 2: Gestión de procesos e hilos

2.1. Concepto de proceso

Un proceso puede ser un **programa en ejecución** o una **instancia** de un programa funcionando en un computador. Es una entidad que puede ser asignada al procesador y ejecutada por él. La entendemos como una **unidad de actividad** caracterizada por:

- La ejecución de una secuencia de instrucciones
- Un estado actual
- Un conjunto de recursos del sistema asociado

En estos procesos nos encontramos una serie de elementos:

- **Código de programa** (puede ser compartido por otros)
- **Datos asociados**
- **BCP** (Bloque de Control de Proceso)
    - PID → Identificador
    - Estado
    - Prioridad para la asignación del procesador
    - Contador de programa
    - Punteros a memoria
    - Datos de contexto (datos que encontramos dentro de los registros del procesador)
    - Información del estado de E/S
    - Información de auditoria

El BCP permite la interrupción de un proceso y su restauración posterior a un estado de ejecución como si no hubiera habido interrupción alguna. Es una **herramienta clave** para que el sistema operativo pueda dar soporte a **múltiples procesos** y proporcionar la **multiprogramación**.

La forma de crear un proceso en Linux es la llamada al sistema *fork()*, por ejemplo:

t_pid pid = fork();

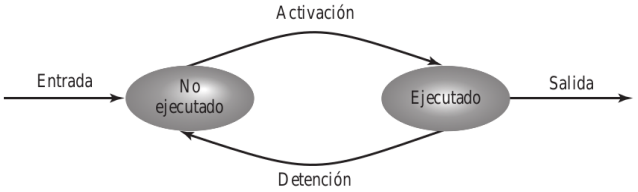
Esta llamada crea un proceso hijo idéntico al proceso padre que la ejecutó. Ambos continúan su ejecución independientemente. La forma de diferenciarlos es por el valor de retorno a la llamada de la función.

Si pid = 0 → Proceso hijo, si pid = 1 → Proceso padre

2.2. Estados de procesos

2.2.1. Modelo de 2 estados

En este modelo es responsabilidad del SO controlar la ejecución de los procesos (patrón entrelazado, asignación de recursos). Consta de 2 posibles estados, **"Ejecutando" y "No ejecutando"**. Cuando el SO genera un nuevo proceso, su BCP consta de estado "No ejecutando" como estado inicial. El proceso en ejecución se interrumpirá de vez en cuando y el activador seleccionará otro proceso a ejecutar.



(a) Diagrama de transiciones de estados

Por tanto:

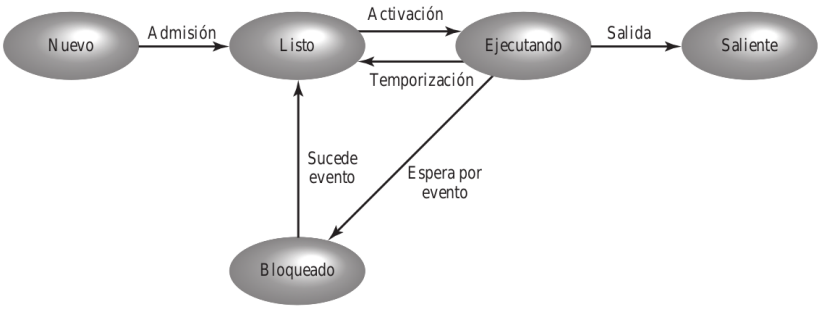
- Proceso saliente: Ejecutando → No ejecutando
- Proceso nuevo: No ejecutando → Ejecutando

En el BCP además del estado actual del proceso, tendremos su localización en memoria (reservada al crear el proceso). Los procesos que no estén en ejecución pueden estar esperando su turno de activación en una cola. Cada entrada de la cola es un puntero al BCP de un proceso.

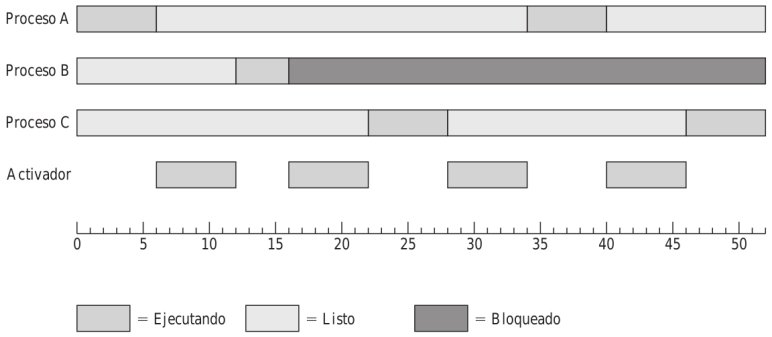
2.2.2. Modelos de 5 estados

Es similar al modelo de 2 estados pero con más estados.

- **Ejecutando**: Proceso actualmente en ejecución.
- **Listo**: Preparado para ser ejecutando cuando tenga oportunidad.
- **Bloqueado**: No se puede ejecutar hasta que se cumpla un evento determinado o se complete la operación E/S.
- **Nuevo**: Proceso recién generado, con su BCP, pero aún no cargado en memoria.
- **Saliente**: Liberado por el SO del grupo de procesos ejecutables debido a haber sido detenido o abortado.

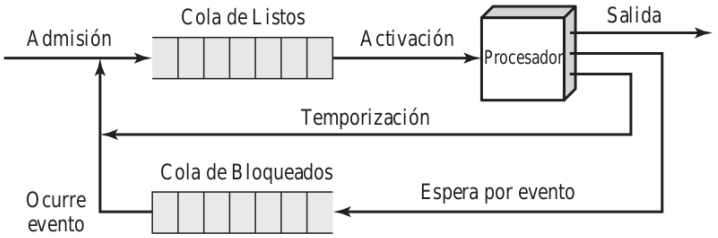


Otro ejemplo:



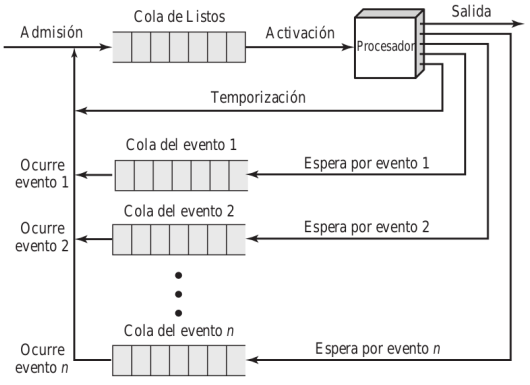
Este modelo de 5 estados se puede encontrar implementado con un **esquema de 2 colas**:

- Cola de **Listos**
  - Cada proceso admitido por el sistema se coloca aquí
  - Si no hay prioridades se trata como una FIFO, sino habría una cola de listos por nivel de prioridad
- Cola de **Bloqueados**
  - Procesos a la espera de un evento
  - Cuando sucede un evento, cualquier proceso de esta cola que espere por ese evento pasa a la cola de Listos.



(a) Cola simple de Bloqueados

También se puede implementar con un esquema de **múltiples colas de Bloqueados**. Con una cola por evento (o dispositivo) se evita que, al suceder un evento, el SO tenga que recorrer la cola entera de Bloqueados para ver cuáles esperan por ese evento. Al suceder un evento, la cola entera pasaría a Listos.



(b) Múltiples colas de Bloqueados

2.2.3. Razones para la creación de un proceso

- **Nuevo proceso de lotes**
  - En un entorno por lotes, un proceso se crea como respuesta a una solicitud de trabajo
- **Sesión interactiva**
  - En un entorno interactivo, un proceso se crea cuando un usuario desde el terminal entra en el sistema
- **Creado por el SO** para proporcionar un servicio
  - El SO puede crear un proceso a petición de una aplicación
    - Ejemplo: Cuando el usuario solicita una impresión de un fichero
- **Creado por un proceso existente** (padre/hijo)
  - Por motivos de modularidad o para explotar la concurrencia, un programa de usuarios puede ordenar la creación de un número de procesos

2.2.4. Razones para la finalización de un proceso

- **Finalización normal**
  - El proceso ejecuta una llamada al SO para indicar que ha completado su ejecución
- **Límite de tiempo excedido**
  - El proceso ha ejecutado más tiempo del especificado en un límite máximo.
- **Memoria no disponible**
  - El proceso requiere más memoria de la que el sistema puede proporcionar.
- **Violaciones de frontera**
  - El proceso trata de acceder a una posición de memoria a la cual no tiene acceso permitido.
- **Error de protección**
  - El proceso trata de usar un recurso (fichero) al que no tiene permitido acceder, o trata de utilizarlo de una forma no apropiada
- **Error aritmético**
  - El proceso trata de realizar una operación de cálculo no permitida (división por 0), o trata de almacenar números mayores de los que la representación hardware puede codificar.
- **Límite de tiempo**
  - El proceso ha esperado más tiempo que el especificado en un valor máximo para que se cumpla un determinado evento.
- **Fallo de E/S**
  - Se ha producido un error durante una operación de E/S, fallo en la L/E después de un límite máximo de intentos, o una operación inválida
- I**nstrucción no válida**
  - El proceso intenta ejecutar una instrucción inexistente
- **Instrucción privilegiada**
  - El proceso intenta usar una instrucción reservada al SO.

- **Uso inapropiado de datos**
    - Una porción de datos es de tipo erróneo o no se encuentra inicializada.
- **Uso inapropiado del operador** o SO
    - El operador o el SO ha finalizado el proceso
- **Finalización del proceso padre**
    - Cuando un proceso padre termina, el SO puede automáticamente finalizar todos sus procesos hijos.
- **Solicitud del proceso padre**
    - Un proceso padre habitualmente tiene autoridad para finalizar sus propios procesos descendientes.

2.2.5. Procesos suspendidos

- Las operaciones de E/S son mucho más lentas.
- Es posible que todos los procesos estén pendientes de E/S
- Necesidad de intercambio (**swap**)
    - Implica mover parte o todo un proceso de memoria principal a disco.
    - Cuando ningún proceso en memoria principal se encuentra en estado Listo, el SO intercambia uno de los procesos bloqueados a disco (cola de Suspendidos).

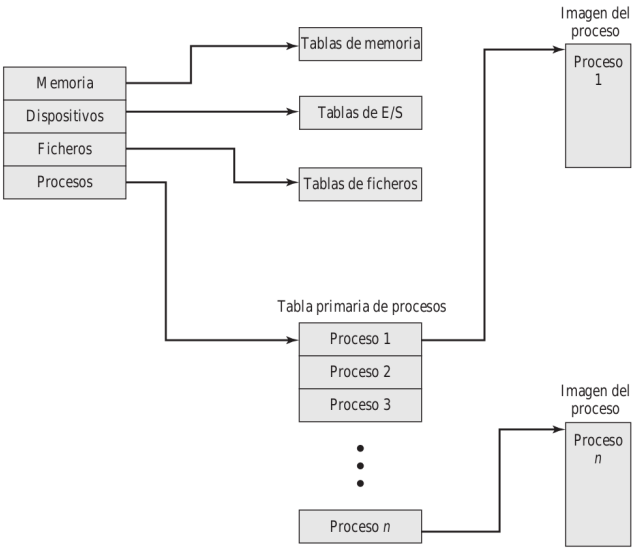
El SO trae a otro proceso de la cola de Suspendidos o responde la solicitud de un nuevo proceso.

2.2.6. Razones para la suspensión de un proceso

- **Swapping**
    - El sistema operativo necesita liberar suficiente memoria principal para traer un proceso en estado Listo de ejecución.
- **Otras razones** del sistema operativo
    - El sistema operativo puede suspender un proceso en segundo plano o de utilidad o un proceso que se sospecha puede causar algún problema.
- **Solicitud interactiva del usuario**
    - Un usuario puede desear suspender la ejecución de un programa con motivo de su depuración o porque está utilizando un recurso.
- **Temporización**
    - Un proceso puede ejecutarse periódicamente (por ejemplo, un proceso monitor de estadísticas sobre el sistema) y puede suspenderse mientras espera el siguiente intervalo de ejecución.
- **Solicitud del proceso padre**
    - Un proceso padre puede querer suspender la ejecución de un descendiente para examinar o modificar dicho proceso suspendido, o para coordinar la actividad de varios procesos descendientes.

2.3. Descripción de procesos

Los procesos constan de una estructura general que se puede analizar con las tablas de control del sistema operativo.



Estos procesos también tienen una imagen que podemos explicar de la siguiente forma:

- **Datos del usuario**
- **Programa del usuario**
    - Programa a ejecutar
- **Pila del sistema**
    - Cada proceso tiene al menos una pila de sistema (LIFO) asociada. Contiene variables locales, parámetros, direcciones de retorno de los procedimientos y llamadas a sistema
- **BCP**: Contiene la información necesaria para que el sistema operativo pueda controlar los procesos y gestionar sus recursos.
    - Identificación del proceso
        - **PID**: Puede ser un índice en la tabla de procesos principal
        - PPID: Identificador del proceso padre
        - Identificador del usuario
    - Información del estado de la CPU
        - **Registros visibles** por el usuario
            - Aquellos a los que se puede hacer referencia por medio del lenguaje máquina que ejecuta la CPU cuando están en modo usuario
        - **Registros de estado y control**
            - Contador de programa
            - Códigos de condición: Resultan de la operación lógica o aritmética más reciente
            - Informacion el estaod
        - Puntero de pila
            - Puntero a la cima de la pila del proceso en ejecución
        - **PSW** (*Program Status Word*)
            - Registro o conjunto de registros que contiene códigos de condición e información de estado

- Información de control de proceso
  - **Información de estado y planificación**
    - Estado del proceso
    - Prioridad
    - Información relativa a la planificación
    - Evento (por el cual se espera)
  - **Estructura de datos**
    - Información estructural, incluyendo punteros que permiten enlazar BCP entre si
      - Ejemplo: Colas de procesos en espera por niveles de prioridad
  - **Comunicación entre procesos**
    - Flags, señales y mensajes relativos a la comunicación entre procesos
  - **Privilegios de proceso**
    - De acuerdo con la memoria a la que van a acceder y los tipos de instrucciones que se pueden ejecutar. También para usar utilidades de sistema o servicio
  - **Gestión de memoria**
    - Punteros a tablas de segmentos y/o páginas de memoria
  - **Propia de recursos y utilización**
    - Recursos controlados por el proceso. También puede incluir un histórico de uso del procesador para el planificador

2.4. Control de procesos

2.4.1. Modos de ejecución

Se necesita proteger al SO y las tablas clave del sistema, de la interferencia con programas de usuario.

- Modo **usuario**
  - Los programas de usuario típicamente se ejecutan en ese modo
- Modo **núcleo**
  - El software tiene control completo del procesador y de sus instrucciones, registros y memoria
- **Funciones** núcleo del SO
  - Gestión de **procesos**
    - Creación y terminación de procesos
    - Planificación y activación de procesos
    - Intercambio de procesos
    - Sincronización de procesos y soporte para comunicación entre procesos
    - Gestión de los bloques de control de proceso
  - Gestión de **memoria**
    - Reserva de espacio de direcciones para los procesos
    - Swapping
    - Gestión de páginas y segmentos

- Gestión de **E/S**
  - Gestión de buffers
  - Reserva de canales de E/S y de dispositivos para los procesos
- Funciones de **soporte**
  - Gestión de interrupciones
  - Auditoria
  - Monitorización

La CPU conoce el modo de ejecución leyendo u bit de la PWS que indica el modo de ejecución del mismo. El nit pasa a modo núcleo (nivel 0) cuando un usuario realiza una llamada a un servicio del SO (*system call*) o cuando una interrupción dispara la ejecución de una subrutina de SO. Al finalizar, se restaura a modo usuario.

2.4.2. Creación de procesos

1. **Asignar un identificador** de proceso único al proceso. Se añade una nueva entrada a la tabla primaria de procesos que contiene una entrada por proceso.
2. **Reservar espacio** para todos los elementos de la imagen del proceso. El SO debe conocer cuanta memoria se requiere para el espacio de direcciones privado y el *stack*.
   a. Asignación por defecto basada en el tipo de proceso
   b. Asignación en base a la solicitud de creación del trabajo remitido por el usuario
   c. Asignación indicada por el proceso padre

   Si existe una parte del espacio de direcciones compartido por este nuevo proceso, se fijan en los enlaces apropiados.

3. **Inicialización del bloque de control de proceso**
   a. Ejemplo: Ka prioridad por defecto se puede fijar la más baja
4. **Establecer los enlaces apropiados**
   a. Ejemplo: Colas de planificador
5. **Creación o expansión de otras estructuras de datos**
   a. Ejemplo: Registro auditoria para análisis del rendimiento del sistema

2.4.3. Cambio de proceso

Ocurre en cualquier instante en el que el SO obtiene el control del proceso en ejecución.

Mecanismo	Causa	Uso
Interrupción	Externa a la ejecución del proceso actualmente en ejecución	Reacción ante un evento externo asíncrono
Trap	Asociada a la ejecución de la instrucción actual	Manejo de una condición de error o de excepción
Llamada al sistema	Solicitud explícita	Llamada a una función del SO

- Cambio de modo

  Si hay una interrupción pendiente, el procesador:

  1. **Asigna al PC** el valor de la dirección de comienzo de la rutina del manejador de interrupción.
  2. **Cambia de modo** usuario a modo supervisor
  3. **Salva el contexto** del programa en ejecución en el BCP del proceso interrumpido
     a. Controlador de programa, registros del procesador, información de la pila

  La existencia de una interrupción no implica necesariamente un cambio de proceso

- Cambio de estado

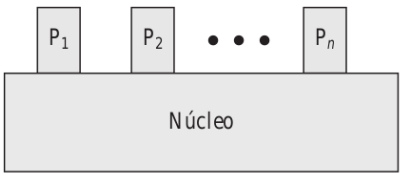
  Si el proceso actualmente se encuentra en estado “Ejecutando” se va a mover a otro estado:

  1. Salvar el **estado del procesador** (PC y registros)
  2. Actualizar el **BCP** del proceso en estado “Ejecutando” al nuevo estado del proceso
  3. Mover el bloque a la **cola apropiada**
  4. Seleccionar **otro proceso** para su ejecución
  5. Actualizar el **BCP** del proceso elegido
  6. Actualizar las estructuras de datos de la gestión de memoria
  7. Restaurar el **contexto del procesador** para el proceso seleccionado

2.4.4. Ejecución del sistema operativo

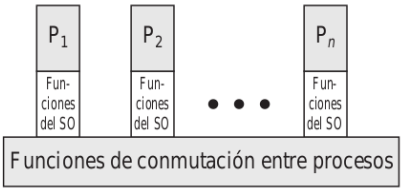
El sistema operativo es un conjunto de programas ejecutados por un procesador. El SO, con frecuencia, cede el control y depende del procesador para recuperar dicho control.

1. Núcleo sin procesos
   a. Ejecución del SO fuera de todo proceso, modo privilegiado
   b. Cuando el proceso en ejecución se interrumpe o invoca una llamada al sistema, el contexto se guarda y el control pasa al núcleo
   c. El SI tiene su propia región de memoria y su propia pila de sistema para controlar la llamada a procedimientos y su retorno
   d. El concepto de proceso se aplica únicamente a los programas de usuario
   e. SO antiguos



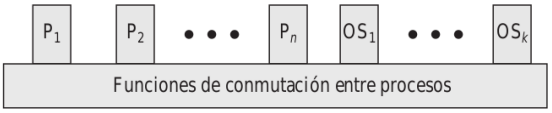
(a) Núcleo independiente

2. Ejecución dentro de los procesos de usuario
    a. Ejecutar virtualmente todo el software del sistema operativo en el contexto de un proceso de usuario
    b. El SO se percibe como un conjunto de rutinas que el usuario invoca para realizar diferentes funciones, ejecutadas dentro del entorno del proceso de usuario
    c. SO de máquinas pequeñas



(b) Funciones del SO se ejecutan dentro del proceso de usuario

3. Sistemas operativos basados en procesos
    a. Sistema operativo implementado como colección de procesos de sistema
    b. Las principales funciones del núcleo se organizan como procesos independientes



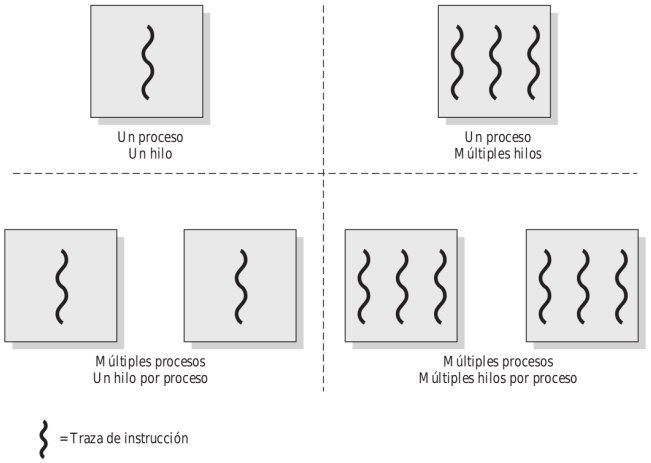
(c) Funciones del SO se ejecutan como procesos independiente

2.5. Hilos

2.5.1. Multihilos

Lo definimos como la capacidad de un SO de dar soporte a múltiples hilos de ejecución en un solo proceso.

- Unidad de propiedad de los recursos (proceso)
- Unidad de expedición (hilo)



- **Proceso**: Unidad de asignación y protección de los recursos
  - Espacio de direcciones virtuales
  - Acceso protegido a los procesadores, otros procesos (comunicación), archivos y recursos E/S

- **Elementos** del hilo. Cada hilo de un proceso contiene:
  - **Estado** de ejecución del hilo
  - **Contexto** del procesador (contador de programa independiente)
  - **Pila** de ejecución
  - Almacenamiento estático para **variables locales**
  - Acceso a **memoria** y a **recursos del proceso** (compartido con el resto de hilos del proceso)

Todos los hilos de un proceso comparten el **estado** y los **recursos** de ese proceso. Residen en el **mismo espacio de direcciones** y tienen acceso a los **mismos datos**.

- Beneficios
  - Lleva menos tiempo crear un nuevo hilo en un proceso existente que crear un proceso totalmente nuevo
  - Lleva menos tiempo finalizar un hilo que un proceso
  - Lleva menos tiempo cambiar entre dos hilos dentro del mismo proceso
  - Se pueden comunicar entre ellos sin necesidad de invocar al núcleo

- Ejemplos de uso
    - **Trabajo en primer y segundo plano**
    - **Procesamiento asíncrono**
    - **Velocidad de ejecución**
        - En un sistema multiprocesador aunque un hilo pueda estar bloqueado por una operación de E/S mientras lee datos, otro hilo puede estar ejecutando
    - **Estructura modular de programas**
        - Un hilo para cada función o fuente de E/S

2.5.2. Funcionalidades del hilo

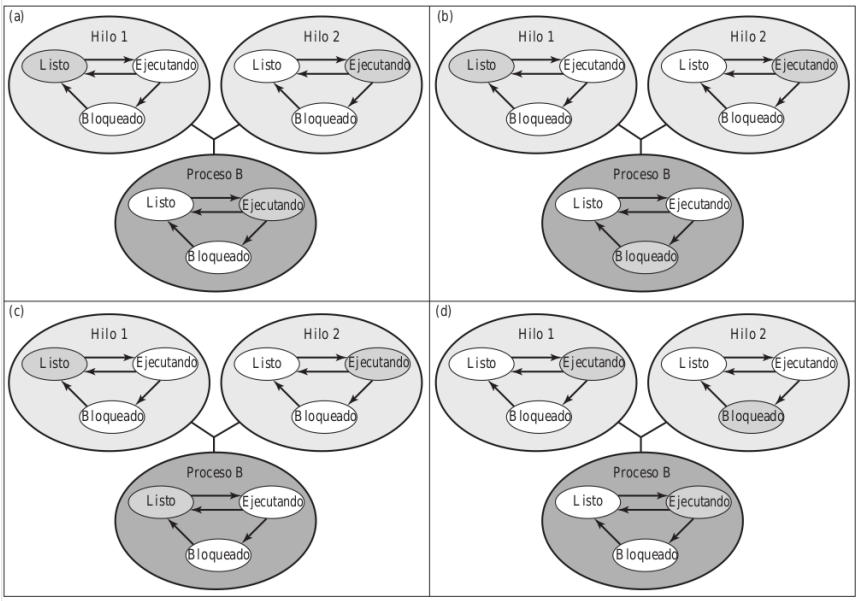
- Estados
    - Ejecutando, Listo y Bloqueado
    - Si se expulsa un proceso, también se expulsan sus hilos, ya que comparten el mismo espacio de direcciones
    - Operaciones asociadas a un **cambio de estado** de hilo:
        - **Creación**
            - Al crearse un proceso también se crea un hilo
            - Un hilo del proceso puede crear otro hilo dentro del mismo proceso, proporcionando un puntero a las instrucciones y argumentos para el nuevo hilo
            - Al nuevo hilo se le dota de su propio registro de contexto, espacio de pila y se coloca en la cola de Listos
        - **Bloqueo**
            - Almacenar registros de usuario, PC y punteros de pila
        - **Desbloqueo**
            - El hilo pasa a cola de Listos
        - **Finalización**
            - Se libera su registro de contexto y pilas

- Sincronización

  Todos los hilos de un proceso comparten el **mismo espacio de direcciones y recursos**, como por ejemplo los archivos abiertos. Cualquier alteración de un recurso por cualquiera de los hilos, afecta al entorno del resto de hilos del mismo proceso. Por eso es **necesario sincronizar las actividades** de los hilos para que no interfieran entre ellos o corrompan estructuras de datos.

- Hilos a nivel de usuario (**ULT**)

  La **aplicación gestiona todo el trabajo de los hilos** y el núcleo no es consciente de la existencia de los mismos. Cualquier aplicación puede **programarse para ser multihilo** usando código de una **biblioteca de hilos**.
    - **Creación y destrucción** de hilos
    - Paso de **mensajes y datos** entre hilos
    - **Planificación** de la ejecución de los hilos
    - **Guardar y restaurar** el contexto de los hilos



- Ventajas
  - El cambio de hilo no requiere privilegios de modo núcleo (ahorra la sobrecarga de 2 cambios de modo)
  - El algoritmo de planificación se puede hacer a medida sin tocar el planificador del SO
  - Los ULT se pueden ejecutar en cualquier SO
- Desventajas
  - Cuando un ULT realiza una llamada al sistema se bloquea u él y todos los hilos del proceso.
    - Solución: Técnica *Jacketing*: El objetivo de esta técnica es convertir una llamada al sistema bloqueante en una llamada al sistema no bloqueante.
  - No se puede aprovechar más de un procesador

- Hilos a nivel de núcleo (**KLT**)

  El **núcleo gestiona todo el trabajo** de gestión de los hilos. No hay código de gestión de hilos en la aplicación, solo una API para acceder a las utilidades del núcleo
  - Ejemplo: Windows
  - Ventajas
    - El núcleo puede planificar simultáneamente múltiples hilos de un solo proceso en un multiprocesador
    - Si se bloquea un hilo de un proceso, el núcleo puede planificar otro hilo del mismo proceso
    - Las rutinas del núcleo pueden ser en sí mismas multihilo
  - Desventajas
    - La transferencia de control de un hilo a otro del mismo proceso requiere un c**ambio de modo** al núcleo

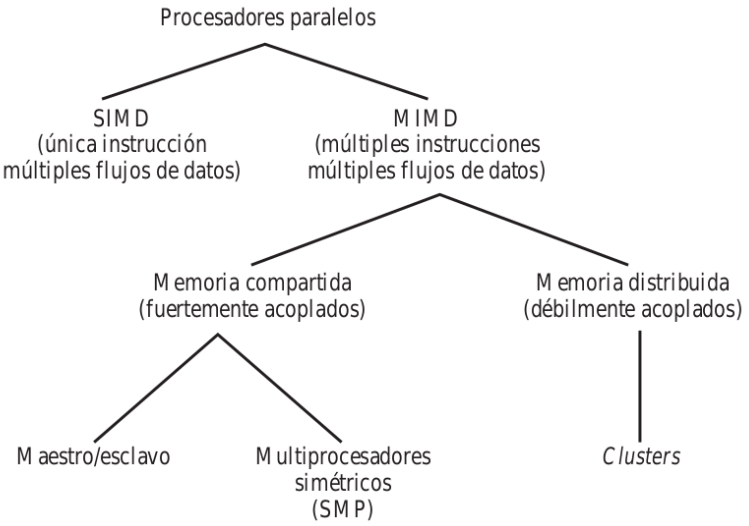
- Enfoques combinados

  La creación de hilos se realiza por completo en el **espacio de usuario**, como la mayor parte de la planificación y sincronización de hilos dentro de una aplicación. Los múltiples **ULT** de una aplicación se **asocian** en un **número menor o igual de KLT.** Por tanto, el programador debe ajustar el número de KLT para una máquina y aplicación en particular para lograr los mejores resultados posibles.

  Añadir que múltiples hilos de la misma aplicación se pueden ejecutar en paralelo en múltiples procesadores y una llamada bloqueante no necesita bloquear el proceso completo.

2.6. SMP

- Arquitectura de procesador paralelo



- Organización del SMP
  - Existen múltiples procesadores, cada uno con su Unidad de Control, ALU y registros
  - Cada procesador tiene acceso a una memoria principal compartida y dispositivos de E/S. El bus compartido es común a todos los procesadores
  - Los procesadores suelen tener al menos un nivel de caché local

- Diseño de SSOO con SMP

  El usuario puede ver el sistema de la misma forma que si fuera un sistema uniprocesador multiprogramado.
  - **Procesos o hilos simultáneos concurrentes**
    - Múltiples procesadores pueden ejecutar la misma o diferentes partes del código del núcleo → Gestión de tablas y estructuras del núcleo
  - **Planificación**
    - Se pueden planificar múltiples hilos del mismo proceso simultáneamente en múltiples procesadores

- **Sincronización**
  - Exclusión mutua y orden de eventos
- **Gestión de memoria**
  - Mecanismos de paginación de los diferentes procesadores coordinados
- **Fiabilidad y tolerancia a fallos**
  - No degrada si falla un procesador

2.7. Micronúcleos

- Solo las funciones absolutamente **esenciales** del SO están en el núcleo.
- Los servicios y aplicaciones menos esenciales se construyen sobre el micronúcleo como servidores de procesos y se ejecutan en modo usuario.
- Cada uno de los servidores puede mandar mensajes al resto y puede invocar **funciones primitivas del micronúcleo**.
- El micronúcleo funcionan como un intercambiador de mensajes y realiza una función de protección.

Tema 3: Concurrencia

3.1. Programas concurrentes

Son programas compuestos por **módulos** que responden a diferentes **eventos**. Cada evento se ejecuta como un proceso **independiente**, **asíncrono** y es responsable de **tareas específicas**. Por complejidad de interacciones y eventos es imposible predecir o repetir una secuencia de ejecución particular → **No determinismo**

Pero estos programas concurrentes pueden tener una serie de problemáticas.

- Errores ocasionados por el **acceso no controlado a recursos compartidos**.
    - Problema de exclusión mutua (o secciones críticas)
    - Son difíciles de detectar y depurar (no determinismo)
- El **intercalado de instrucciones** puede producir inconsistencia secuencial
    - Independientemente del número de procesadores
- La **velocidad relativa** de ejecución de los procesos no puede ser predecidla
    - Depende de las actividades de otros procesos
    - La manera que el SO maneja las interrupciones
    - Estrategias de planificación

3.1.1. Condición de carrera

Es cuando múltiples procesos o hilos leen y escriben en **objetos compartidos** y el resultado final depende del **instante** y **orden** en que cada uno ejecuta las instrucciones.

- El “perdedor” es el que modifica el último y determinará el valor final de la variable
- Errores difíciles de detectar y de repetir debido al planificador **no determinista** (diferentes intercalados en cada ejecución)

3.1.2. Operaciones atómicas

Es cuando se ejecutan una secuencia de instrucciones que son indivisibles. Se ejecutan todas como un grupo o ninguna se ejecuta. El proceso no puede interrumpirse y ejecutarse otro que modifica esas mismas variables y hay operaciones básicas que no son atómicas, donde el código ejecutable está compuesto por varias instrucciones de procesador.

3.2. Exclusión mutua

Se trata de un problema básico y fundamental de sincronización entre procesos con memoria compartida (o hilos). Por tanto, se debe asegurar el acceso ordenador a los recursos compartidos para **impedir errores e inconsistencias**.

3.2.1. Sección crítica

Es la parte del código que accede, modifica recursos o zonas de **memoria compartidas**, ejecutado por un conjunto de procesos independientes que pueden ser considerados cíclicos. La intercalación de instrucciones en esas secciones críticas provocan **condiciones de carrera** que pueden generar **resultados erróneos** dependiendo de la secuencia de ejecución.

- **Modelo** de sección crítica
  - Asegurar la **exclusión mutua** de la ejecución de las secciones críticas
    - Mientras se ejecuta una de ellas no se debe permitir la ejecución de las de otros procesos
  - El modelo separa el código en secciones críticas y resto del código
  - La solución se basa en desarrollar los **algoritmos** que se insertan justo antes (**preprotocolo**) y después (**postprotocolo**) de las secciones críticas

```
while True:
    ...
    entry_critical_section()
    critical section()
    exit_critical_section()
```

3.3. Soluciones por software: algoritmos de exclusión mutua

Se tienen que cumplir tres requisitos:

- Asegurar la **exclusión mutua**
  - Asegurar que solo uno de los procesos ejecuta instrucciones de la sección crítica
- Progreso o **libre de interbloqueos** (*deadlock*)
  - Si varios procesos desean entrar a la sección crítica, al menos uno de ellos sabe poder hacerlo
- **Espera limitada** o libre inanición (*starvation*)
  - Cualquier proceso debe poder entrar a la sección crítica en tiempo finito (no siempre se puede asegurar)

Sin embargo, con el algoritmo de Dijkstra habla de cuatro requisitos:

- La solución debe ser **simétrica**
  - No se permiten soluciones que cambien el comportamiento o la prioridad estática de algún proceso
- No se deben hacer suposiciones de la **velocidad relativa** de los procesos, ni se puede suponer que las velocidades sean constantes
- Entrada inmediata o **no interferencia**
  - Un proceso que se interrumpe, en el resto de código no debe interferir ni bloquear a los demás procesos
- Si varios procesos desean entrar simultáneamente, la decisión en la entrada de la sección crítica debe tomar un **número finito de pasos**

3.3.1. Solución para 2 procesos

- Primer intento
  - ✓ **Asegurar la exclusión mutua**: La variable turn solo puede tomar uno de los 2 valores
  - ✗ **Progreso**: La entrada de P0 está interferida por P1 cuando este no tiene intención de entrar o no se está ejecutando
  - ✗ **Espera limitada**: Espera infinita si P1 no entra en la sección crítica
  - ✓ **Solución simétrica**: Mismo código P0 y P1 con valores intercambiados

    - ✗ **Sin suposiciones de velocidad relativa**: Hemos supuesto que P0 y P1 entraran alternativamente
    - ✗ **Entrada inmediata**: Si *turn* = *1*, pero P1 está en el resto del código, P0 no podrá entrar
    - **Problema**: Obliga a la alternancia estricta, procesos lentos atrasan los rápidos
- Segundo intento
    - Cuando un proceso desea entrar verifica el estado del otro. Si no está en la sección crítica pone *true* en su posición del array y continua (entrando en la sección crítica)
    - **Problema**: No asegura exclusión mutua (ambos valores de *states* pueden ser *true*)
- Tercer intento
    - Si se cambia el estado propio antes de verificar el del otro se impedirá que los dos entren simultáneamente en la sección crítica
    - ✓ **Asegurar la exclusión mutua**: Si hay competencia, el primero que ejecute la asignación a *states* entrará en la sección crítica
    - ✓ **Entrada inmediata**: Si P1 está en el resto del código, entonces *states[1]* será falso, por lo que no interfiere con P0 y este podrá entrar y salir varias veces sin esperas
    - ✗ **Progreso**: Interbloqueo (*deadlock*)
    - **Problema**: Interbloqueo
- Cuarto intento
    - Se abre una ventana temporal para que uno de los procesos pueda continuar
    - ✓ **Asegurar la exclusión mutua**. Hay dos casos:
        - Primero, P0 entra en su sección crítica antes de que P1 verifique el calor de *states[0]* → P1 quedará esperando
        - Segundo, hay competencia entre procesos que entran en el bucle. Para que uno pueda salir, por ejemplo P0, entonces P1 debe interrumpirse justo después de ejecutar *states[i]* = *false*. P0 podrá continuar y P1 deberá esperar
    - ✓ **Progreso**
    - ✗ **Espera limitada**: En la práctica y estadísticamente no se producen esperad infinitas, pero no se puede asegurar que la espera estará limitada a un número de pasos finitos → **bloqueo activo** (*livelock*). En algún momento uno de ellos saldrá del bloqueo, pero mientras tanto ambos procesos cambian valores de una variable sin hacer nada útil
    - ✓ **Solución simétrica**: Mismo código P0 y P1 con valores intercambiados
    - ✓ **Sin suposiciones de velocidad relativa**
    - ✓ **Entrada inmediata**: Si uno de los procesos no desea entrar en la sección crítica, su estado en *states* será *false* y el otro podrá entrar sin esperar

    - **Problema**: Indefinición dentro del bucle, postergación indefinida
- Algoritmo de Dekker (1963)
    - Si ambos procesos entran en el bucle, el valor de *turn* decidirá qué proceso accede
    - ✓ **Asegurar la exclusión mutua**. Si P1 desea entrar en la sección crítica y P0 ya está en ella, P1 quedará esperando. Cuando P0 salga, pondrá *turn = 1* por lo que el siguiente en entrar será P1, aunque P0 intente volver a entrar inmediatamente
    - ✓ **Progreso**: En caso de competencia ambos procesos verifican el valor de *turn*, uno de ellos (y solo uno) entrará en la sección crítica sin espera adicional
    - ✓ **Espera limitada**: Cuando salga el proceso que haya entrado primero, dará el turno al que quedó esperando, como en el primer caso
    - **El algoritmo es correcto pero se puede simplificar**
- Algoritmo de Peterson (1981)
    - Espera si el estado del otro es *true* y es el turno del otro
    - ✓ **Asegurar la exclusión mutua** 3 casos:
        - *states[false, false]* → Imposible porque los procesos que desean entrar antes asignan *true* a su posición
        - *states[true, true]*, *turn = 0* y que el último que desea entrar sea P0 → Imposible porque antes de la comparación, P0 hizo *turn = 1* (lo mismo se aplica si es P1)
        - Hay competencia y *turn* vale 0 y 1 simultáneamente → Imposible porque el primero que haya ejecutado *turn = x* será él el que entrará
    - ✓ **Progreso**
        - Si hay competencia, el valor de *turn* decide que proceso continúa
        - Si no hay competencia, el proceso que pretende entrar lo hará inmediatamente porque el valor de *states* para el otro será *false*
    - ✓ **Espera limitada**
        - El proceso que desea entrar, primero cede el turno al otro antes de la comparación en el bucle
        - En caso de competencia, el proceso que intenta volver a entrar cederá el turno al que ya estaba esperando
        - Cada proceso espera como máximo un único paso
        - Si hay competencia podrá entrar cuando haya salido el que entró previamente

3.3.2. Solución para N procesos

- Algoritmo de la panadería (1974)
  - ✓ **Asegurar la exclusión mutua**
    - Para que 2 procesos estén en la sección crítica ambos deberían tener el mismo número → El uso del ID único y la relación de precedencia asegura que siembre uno será el menor y será el único que saldrá del último bucle.
    - Para que un 2º proceso (P2) entre en la sección crítica si ya está P1, ha de cumplirse que *number[2]* < *number[1]*. No puede ocurrir:
      - Si P1 salió del bucle de *choosing* es porque P2 ya salió de la selección, por tanto, su número será comparado en el siguiente bucle de comparación de números y habrá entrado P2 antes que P1.
      - Si P2 todavía no entró en la selección, entonces lo hará después de que P1 haya almacenado su número, y por *number[2] = 1 + max(number)* seleccionará un número mayor que el de P1
  - ✓ **Progreso**
    - El peor caso de competencia es que todos los procesos pretenda entrar simultáneamente y hayan seleccionado el mismo número
    - En este caso siempre habrá un único proceso menor que podrá entrar en la sección crítica
    - Cuando salga, podrá entrar el siguiente con el ID más bajo, y así sucesivamente en el orden de los IDs
  - ✓ **Espera limitada**
    - Si un proceso sale de la sección crítica y pretende volver a entrar, cogerá un número mayor de los que ya están esperando; por lo que esos entrarán antes
    - Si N procesos desean entrar simultáneamente, como máximo tendrán que esperar que entren otros N-1 procesos
    - **Es equitativo**: Todos los procesos entran en el orden de número
  - **Problema**: No asegura la exclusión mutua
  - **Solución**:
    - Usar el ID de cada proceso para desempatar en caso de que hayan seleccionado el mismo número
    - Impedir que un proceso avance si el proceso, contra el que está por comparar su número, todavía lo está seleccionando

3.4. Soluciones por hardware

- Problemas
  - *Spinlocks* (**espera activa**)
  - El número de registros necesarios (tamaño de arrays) crece linealmente con el número de procesos concurrentes
    - En sistemas SMP → Sobrecarga mantenimiento consistencia memoria caché
  - Con único procesador → Avance lento e impredecible dependiente del grado de competencia y comportamiento del scheduler
  - Barreras de memoria para consistencia secuencial

- Deshabilitación de interrupciones
  - Si el problema principal es el intercalado producido por las interrupciones, una solución sería deshabilitarlas
  - Se recurre a ello en casos muy concretos
  - No aceptable para procesos de usuario
    - Si un proceso puede deshabilitar las interrupciones también puede tomar el control del sistema → Se anula la cualidad de apropiativo del *scheduler*
  - Procesos del núcleo del sistema
    - Dificultad de deshabilitar interrupciones simultáneamente en todos los procesadores → No se previene el efecto de intercalación

3.4.1. Test & Set

- Solución más usada hasta la década de 1970
- La implementación en hardware usa una variable entera binaria (o booleana) que puede tomar valores 0 y 1
- La instrucción **atómica** retorna el valor del registro
  - Si su valor es 0, le asigna 1 y retorna 0
  - Si su valor es 1, retorna 1

```
def TAS(register):
  if register == 0:
    register = 1
    return 0
  return 1
```

3.4.2. Swap

- Intercambia atómicamente 2 posiciones de memoria (palabras de 32 o 64 bits)
- Muy parecido a TAS solo que el valor anterior de *mutex* se verifica en la variable local de intercambio

```
def swap (register1, register2):
  tmp = register1
  register1 = register2
  register 2 = tmp
```

3.4.3. Compare & Swap

- Requiere **3 operandos**:
  - Registro (*register*): Valor a comparar
  - Valor esperado (*expected*)
  - Nuevo valor (*desired*): Se asignará al registro si su valor era el esperado

```
def CAS (register, expected, desired):
  if register == expected:
    register = desired
    return True
  else
    expected = register
    return False
```

En estas soluciones por hardware sigue habiendo una espera activa. Mientras un proceso está esperando para acceder a una sección crítica, continúa consumiendo tiempo de procesador. Tampoco podemos asegurar espera limitada. Cuando un proceso abandona su sección crítica y hay más de un proceso esperando, la selección del proceso en espera es arbitraria. Así, a algún proceso podría denegársele indefinidamente el acceso.

3.5. Mecanismos de sincronización del SO

3.5.1. Semáforos

Este tipo de mecanismo fueron inventados por el matemático Dijkstra en la década de 1960, con la finalidad de solucionar la problemática de sincronización sin espera activa de los algoritmos concurrentes. Inspirado en las señales visuales ferroviarias que indican si un tren puede entrar en una vía o no. Los procesos se bloquean o ejecutan condicionados únicamente por el valor que tiene una variable entera. Se implementan habitualmente como servicios de núcleo de los SSOO.

Un semáforo es una construcción definida por:

```
struct Semaphore {
  unsigned value;
  Queue queue;
}
```

- **Una variable entera**: El valor del semáforo (consideramos valores no negativos)
- **Cola de procesos bloqueados en el semáforo** (para evitar la espera activa)
- Podemos inicializar el valor *value* a 1
- Además, se componen 2 primitivas **atómicas** y **mutuamente exclusivas**:
    - Si el valor del semáforo es mayor que 0, se decrementa su valor. Si es cero, se bloquea al proceso que llamó a *wait* y se le añade a la cola de procesos bloqueados por el semáforo

```
def wait(s):
  if s.value > 0:
    s.value -= 1
  else:
    add(process, s.queue)
    bloc(process)
```

    - Si no hay procesos bloqueados en la cola del semáforo, se incrementa el valor del semáforo. En caso contrario se desbloquea un proceso y pasa a poder ser seleccionado para ejecución

```
def signal(s):
  if empty(s.queue):
    s.value += 1
  else:
    process = get(s.queue)
    sched(process)
```

- **Exclusión mutua**
  - Valores **positivos**

    Cuando un primer proceso entre en la sección crítica, decrementará el valor del semáforo y continuará. Si otro proceso desea entrar, el valor del semáforo será 0 y, por tanto, se bloqueará hasta que el primero ejecute el *signal*.
  - Valores **negativos**
    - Decrementa el valor del semáforo. Si el valor es negativo, el proceso se bloquea y se añade a la cola, sino sigue su curso

```
def wait(s):
  if s.value > 0:
    s.value -= 1
  else:
    add(process, s.queue)
    bloc(process)
```

    - Incrementa el valor del semáforo. Si el valor ≤ 0 se desbloquea un proceso y se extrae de la cola (FIFO)

```
def signal(s):
  if empty(s.queue):
    s.value += 1
  else:
    process = get(s.queue)
    sched(process)
```

    - Significado de *s.value*:
      - >= 0 → Ese valor indica el número de procesos que pueden ejecutar *wait(s)* sin bloquearse, es decir a cuantos procesos dejamos usar en la sección critica al mismo tiempo
      - < 0 → El valor absoluto de ese número indica en número de procesos bloqueaos en la cola del semáforo

- Semáforos **mutex y locks**

  Son **semáforos binarios** optimizados para ser usados con **exclusión mutua**. Normalmente, se inicializan automáticamente a 1 (para dejar paso al primer proceso), pero en algunas implementaciones, solo el proceso que hizo *lock* puede luego hacer el *unlock* (**proceso propietario**).

  Algunos sistemas permiten que un mismo hilo haga varios *lock*. Si ya es el propietario del mutex continua su ejecución → **Semáforos mutex reentrantes.**

  - Si el valor es 1 → Se cambia a 0. Si el valor es 0 → El proceso se bloquea y es añadido a la cola

```
def lock(mutex):
  if mutex.value == 1:
    mutex.value = 0
  else:
    add(process, mutex.queue)
    bloc(process)
```

  - Si no hay procesos en cola → El valor se pone a 1, sino se desbloquea un proceso de la cola

```
def unlock(mutex):
  if  empty(mutex.queue):
    mutex.value = 1
  else:
    process = get(mutex.queue)
    sched(process)
```

  Hay dos tipos:
  - Semáforo fuerte
    - Cola FIFO
    - Garantiza espera limitada y equidad
  - Semáforo débil
    - Selección de proceso aleatoria

- **Productores-Consumidores**

  Es un ejemplo clásico de sincronización de orden de ejecución. Está presente en mecanismos de comunicación, tuberías entre procesos y comandos, E/S a dispositivos, comunicaciones por red, streaming…

  Encontramos 2 tipos de procesos:
  - **Productor**
    - Produce un nuevo elemento que será transmitido a los consumidores
  - **Consumidor**
    - Recibe y consume los elementos transmitidos desde los productores

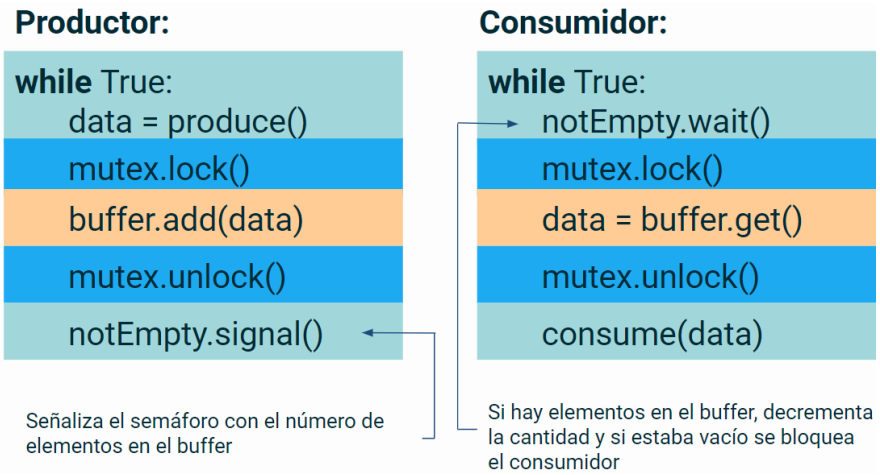
  Encontramos también 2 tipos de productores-consumidores
  - Síncronos
    - Cuando se produce un elemento, este debe ser consumido para que el productor pueda continuar su ejecución
  - Asíncronos
    - Canal de comunicación con capacidad de almacenamiento (*buffer*) → Los productores no han de esperar a que cada elemento sea consumido
    - Los productores agregan los elementos a una cola y los consumidores obtienen el primer elemento de esta
    - Productores y consumidores avanzan a su propio ritmo
    - Requiere cierta sincronización
      - Consumidores han de esperar si el *buffer* está vacío
      - Productores han de esperar si el *buffer* está lleno

  Los procesos pueden ser considerados cíclicos, ambos ejecutan un bucle donde añaden o quitan elementos del *buffer*.

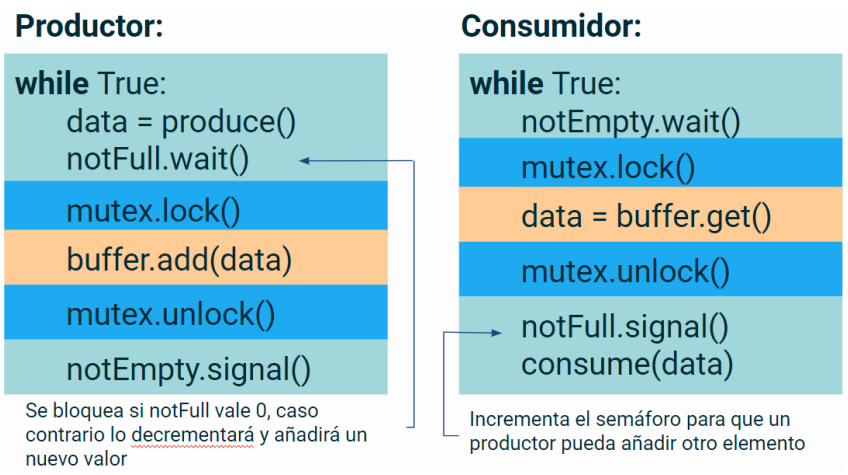
Productor:
```
while True:
    data = produce()
    buffer.add(data)
```

Consumidor:
```
while True:
    data = buffer.get()
    consume(data)
```

  Los accesos al *buffer* van a estar en una sección crítica.
  - 1ª Aproximación: Buffer ~~infinito~~
    - Consumidores bloqueados si el *buffer* está vacío
    - 2 semáforos
      - ***mutex***: PAra asegurar exclusión mutua al *buffer*
      - ***notEmpty***: Semáforo contador de sincronización para bloquear consumidores si el *buffer* está vacío. Su valor es el número de elementos disponibles en el *buffer* para consumir



- 2ª Aproximación: Buffer infinto corcular
  - Consumidores bloqueados si el *buffer* está vacío
  - Productores bloqueados si no quedan posiciones libres
  - 3 semáforos
    - ***mutex***: Para asegurar la exclusión mutua al *buffer*
    - ***notEmpty***: Semáforo contador del número de elementos disponibles en el *buffer* (para bloquear el consumidor)
    - ***notFull***: Semáforo contador del número de posiciones libres (para bloquear productor)



- **Lectores-Escritores**
  - Se basa en la relajación de las condiciones de exclusión mutua
    - Se permite **más de un lector** en la sección crítica
    - Mientras haya **un lector** en la sección crítica no puede entrar **ningún escritor**
    - Los **lectores no pueden entrar si hay un escritor** en la sección crítica
    - Solo puede haber un **escritor** en la sección crítica
  - Implementación con **semáforos binarios** si permiten que un proceso que no hizo el *wait* pueda hacer un *signal*
  - 2 semáforos:

        - ***writer***: Para **asegurar exclusión** mutua entre escritores y entre escritor y lectores
        - ***mx***: Semáforo para:
            - **Asegurar exclusión mutua** para verificar y modificar *readers*
            - **Barrera**: El primer lector bloqueará a los siguientes si hay un escritor en la sección crítica
    - Conclusiones
        - El algoritmo da **prioridad a los lectores**
        - **No asegura espera limitada a los escritores**
        - Cuando entra un lector, los escritores tendrán que esperar hasta que salga el último, pero los lectores podrán seguir entrando sin dejar paso al escritor → **Esperas infinitas**
            - Para evitarlas hay que asegurar que los lectores esperan si un escritor desea entrar
        - 3 semaforos:
            - *writer*
            - *mx*
            - ***entry***: **Bloquea** a los nuevos lectores cuando el primer escritor hace un *wait*

- El **problema de los filósofos**
    - Cada filósofo es un **proceso** que realiza solo dos actividades: **pensar o comer**
    - Cada filósofo necesita **2 palillos** para comer y solo puede tomar los que tiene a su lado

```
def philosopher():
    while True:
        think()
        pick()      <- asegura que puede tomar 2 tenedores (derecha e izquierda)
        eat()
        release()   <- libera ambos tenedores
```

    - Requisitos
        - Un filósofo solo puede comer si tiene **2 palillos**
        - **Exclusión mutua**: Un tenedor solo puede ser usado por un filósofo a la vez
        - Se debe asegurar **progreso**
        - Se debe asegurar **espera limitada**
        - Debe ser **eficiente**
            - Si hay competencia por un palillo, este debe poder ser usado por uno de sus 2 filósofos vecinos
    - Identificamos a los **filósofos y palillos** con un índice de 0 a N-1 (N=5)
        - El tenedor de la izquierda del filósofo$_0$ será el tenedor$_0$, el de su derecha el tenedor$_1$
        - El filósofo$_4$ a su izquierda tendrá el tenedor$_4$ y a su derecha el tenedor$_0$
    - **1ª Solución** asegurando exclusión mutua a toda la mesa: Solo un filósofo puede comer a la vez
        - Ineficiente

    - **2ª Solución** asegurando exclusión mutua por cada tenedor

        - Se necesita un ***array* de 5 semáforos *mutex***, uni por tenedor
        - El índice *i* identifica a cada filósofo, cada proceso intentará tomar primero el tenedor de su izquierda (i) y luego el de su derecha (i+1)%5

Estas soluciones pueden provocar interbloqueo, es por eso que hay que tener en cuenta lo siguiente:

- **Exclusión mutua**
    - Los recursos solo pueden asignarse a un proceso
- **Retención y espera** (*hold and wait*)
    - Un proceso mantiene los recursos ya asignados mientras espera la asignación del otro
- **No apropiación** (*no preemption*)
    - No se puede quitar un recurso que está asignado a un proceso, debe ser este el que lo libere
- **Espera circular** (*circular wait*)
    - Se produce un bucle, procesos esperando por recursos asignados a otros

Por tanto, para encontrar la solución óptima debemos hacer un **cambio de enfoque**. En vez de tratarlo como un problema de exclusión mutua, tenemos que tratarlo como una **sincronización de orden de instrucciones**.

Cuando un filósofo desea comer, **verifica el estado de sus 2 vecinos**. De esta forma podrá comer si ninguno de estos come y en caso contrario, tendrá que esperar a que los 2 le notifiquen cuando liberen los palillos.

- El array *status* indica el estado de cada filósofo
    - Pensando
    - Hambriento
    - Comiendo
- Semáforo *mutex* para asegurar la exclusión mutua cuando se verifica y manipula el array *status*
- Array *sync* de semáforos para sincronizar entre los filósofos

```
def canEat(i):
    if status[i] == HUNGRY
            and status[left(i)] != EATING
            and status[right(i)] != EATING:
        status[i] = EATING
        sync[i].release()
```

Si ninguno está comiendo, *canEat* asigna *EATING* al estado de filósofo, y señaliza su semáforo. Las funciones *left* y *right* retornan el índice del filósofo vecino (no del tenedor).

Cuando un filósofo deja de comer, debe comprobar si sus vecinos están esperando por los tenedores que retenía. Antes de señalizarles también tiene que revisar que el otro vecino de su vecino no está comiendo.

```
def release(i):
    mutex.acquire()
    status[i] = THINKING
    canEat(left(i))
    canEat(right(i))
```

- Para ello se reusa *canEat* con otros argumentos
- Si el filósofo que deja los tenedores es el 1, entonces se llamará con el argumento 0 (filosofo de la izquierda) y luego con el 2 (filosofo de la derecha)

Las llamadas a *canEat* se hacen siempre desde dentro de la sección crítica del semáforo *mutex*

- No hay **condiciones de carrera**
- No hay **conflictos** en las **verificaciones** y **cambios de estado** del array *status*

El algoritmo es **óptimo**, asegura que si hay tenedores para que coman 2 filósofos, estos podrán hacerlo **sin esperar**. Tampoco hay **retención y espera**, los filósofos que no pueden comer no retienen ningún tenedor y, por tanto, no se puede producir una **espera circular**. En consecuencia, el resultado es un **algoritmo sin interbloqueos**.

La **función *pick*** asigna *HUNGRY* al estado del filósofo y llama a la función *canEat*, que verifica si ninguno de los vecinos está comiendo. Si no es así, señaliza en su semáforo sync correspondiente, por lo que no se bloqueará en el *acquire* sobre *sync[i]*. Pero si alguno de los vecinos está comiendo no se hará el *release* y el filósofo se bloqueará.

```
def pick(i):
    mutex.acquire()
    status[i] = HUNGRY
    canEat(i)
    mutex.release()
    sync[i].acquire()
```

3.5.2. Monitores

Los monitores son **estructuras de datos abstractas y primitivas** de programación concurrente que concentran la **responsabilidad** en los módulos de los programas. Son una **generalización del núcleo** de los primeros sistemas operativos y los solemos ver integrados en lenguajes de **alto nivel** (Java).

- **Sistema Operativo**: Conjunto de **módulos** que asigna recursos compartidos para diversos procesos
- **Monitor**: Conjunto de **procedimientos y datos** que debía gestionar cada módulo

Cada monitor debía asegurar la **exclusión mutua** de la ejecución de sus **procedimientos**. Ningún procedimiento de un monitor se ejecutará si otro se está ejecutando.

Las **variables** del monitor solo podían ser accedidas desde estos procedimientos y de esta forma el problema de la **sección crítica** quedaba resuelto.

- Java
  - Java implementa monitores como **construcción sintáctica** del lenguaje
    - Comunicación de **métodos y bloques *synchronized*** con las funciones de sincronización *wait, notify* y *notifyAll*
    - **Monitor Java**: Clase cuyos métodos públicos están todos declarados como *synchronized*
  - Cada **objeto** en Java tiene asociado un ***mutex* implícito**

  - El *mutex* se puede usar para forzar la exclusión mutua de un **bloque de código** indicando que está sincronizado con el **objeto**

```
Object lock = new Object();
...
for (int i=0; i<max; i++) {
  synchronized (lock) {
    counter += 1;
  }
}
```

Java agrega automáticamente las operaciones *lock* y *unlock* sobre el *mutex* del objeto al inicio y salidas del bloque de código.

  - Alternativa equivalente y más simple: **Declarar *synchronized*** a los **métodos** que acceden a recursos compartidos

```
...
  synchronized void add() {
    counter++;
  }
...
    for (int i=0; i<max; i++) {
      add();
    }
```

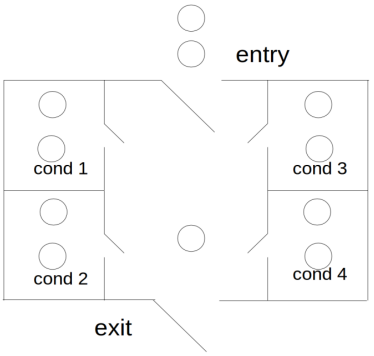
El *mutex* está asociado a la propia instancia, el objeto *this*.
El prefijo *synchronized* especifica que el hilo debe obtener el *lock* para ejecutar el método.
Las llamadas a otros métodos retienen el acceso exclusivo hasta que se haya salido del método *synchronized*.

  - El *mutex* no es global
    - Un método de instancia *synchronized* solo asegura la exclusión mutua de ese método sobre la **misma instancia**.
    - Cada instancia de una clase ejecuta sus métodos **independientemente** de las demás.
    - Para garantizar exclusión mutua si varias instancias modifican concurrentemente una variable estática desde un método de instancia *synchronized*,
      - Definir **objeto estático compartido por las diferentes instancias**
      - Hacerlo desde un **método de clase *synchronized***

- Variables de condición
  La **exclusión mutua** entre **procedimientos** no es suficiente para la sincronización general entre **procesos**. Se añadieron las operaciones:
  - ***wait***
  - ***notify*** (≈ signal)
  - ***notifyAll*** (≈ broadcast: despierta TODOS los hilos en la cola)

  Permiten bloquear y desbloquear procesos cuando se cumple alguna condición (variables de estado).
  - Ej.: Bloquear a los productores si el buffer está lleno y desbloquearlos cuando hay nuevamente espacio.

- Representación gráfica
  - Por la exclusión mutua únicamente **1 proceso** puede estar **dentro** del monitor.
  - Los procesos dentro del monitor pueden bloquearse en **variables de condición** (salas internas) → Liberan temporalmente el *lock* para que otros puedan entrar.
  - Prioridades monitores Java:



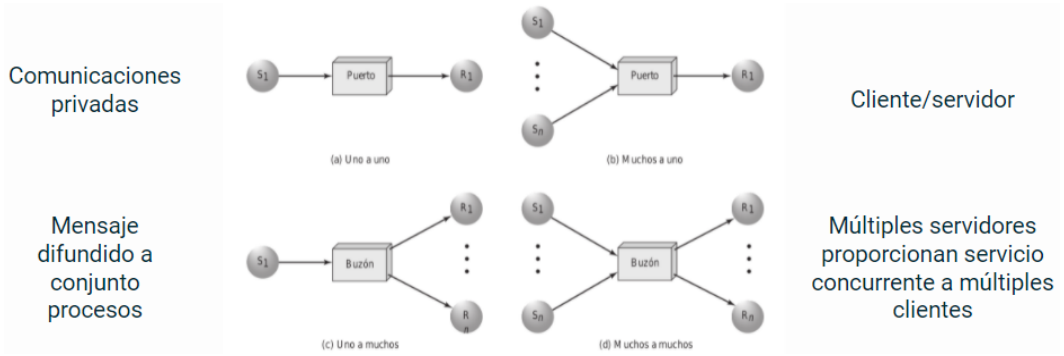
E = W < S

(S: proceso que señaliza,
W: proceso en sala interna,
E: proceso en cola para entrar en el monitor)

3.5.3. Canales

Los canales permiten el paso de mensajes que proporciona las funciones de sincronización y comunicación. Pueden ser implementados en:

- Sistemas distribuidos
- Multiprocesadores de memoria compartida
- Sistemas monoprocesador
- Sentencias **primitivas** (atómicas)
  - send (destino, mensaje)
  - receive (origen, mensaje)
- Direccionamiento
  - Identificando el **proceso**
    - send **(id_proceso_destinatario**, mensaje)
  - Identificando el **canal**



- Características
  - **Tipos de datos** estáticos/dinámicos
  - **Entrega de mensajes** por orden de envío (FIFO)/en orden arbitrario
  - **Asegurar la recepción** de cada mensaje (*reliable*) o no
  - **Descartar mensajes** por errores de transmisión (*best-effort*) o no
- Tipos de comunicación
  - **Síncrona** (*rendevouz*)
    - El **remitente se bloquea** en el *send* hasta que el receptor ejecuta el *receive*
    - El **receptor se bloquea** hasta que el remitente envía el mensaje
  - **Asíncrona** → *buffer* finito (*mailbox*)
    - El **remitente no se bloquea** (prosigue su ejecución sin esperar que el receptor lo reciba) a menos que se llene el buffer
    - El receptor generalmente se bloquea

Tema 4: Planificación de procesos

4.1. Objetivos de la planificación

Los objetivos que nos encontramos son:

- Equidad
- Ausencia de inanición de cualquier proceso
- Uso eficiente del tiempo del procesador
- Poca sobrecarga
- Niveles de prioridad o plazos de tiempo real para el inicio o finalización de ciertos procesos

4.2. Tipos de planificación

A largo plazo	Decisión de añadir un proceso al conjunto de procesos a ser ejecutados (cuando se crea el proceso). Establece el nivel de multiprogramación
A medio plazo	Decisión de añadir un proceso al número de procesos que están parcialmente o totalmente en la memoria principal (swapping)
A corto plazo	División de cuál de los procesos están listos se elige para ser ejecutado por el procesador
De E/S	Decisión por la que un proceso que está pendiente de una petición de E/S será atendido por un dispositivo de E/S disponible

4.3. Planificación a corto plazo

- Evento → Bloqueo del proceso, selección de otro a ejecutar
- Ejemplos de eventos:
  - Interrupciones de reloj (SO apropiativo)
  - Interrupciones de E/S
  - Llamada al SO
  - Señales

4.4. Criterios de planificación

4.4.1. Orientados al usuario

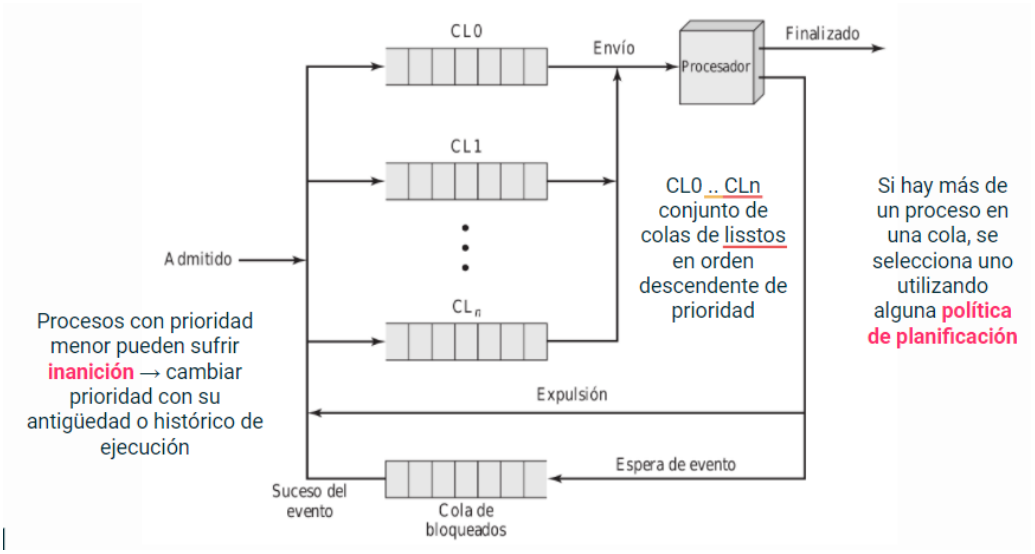
- Relacionados con prestaciones
  - Tiempo de estancia o retorno (***tournaround time***)
    - Tiempo transcurrido desde que se lanza un proceso hasta que finaliza. Incluye el **tiempo de ejecución** sumado con el **tiempo de espera** por los recursos, incluyendo el procesador
  - Tiempo de respuesta (***response time***)
    - Para un proceso interactivo, es el tiempo que transcurre desde que se lanza una petición hasta que se comienza a recibir la respuesta.
    - Más importante para el usuario

- Caducidad (***deadline***)
  - Subordinar otros objetivos al de maximizar el porcentaje de fechas tope conseguidas
- Otros
  - Previsibilidad
    - Un trabajo dado debería ejecutarse aproximadamente en el **mismo tiempo** y con el **mismo coste** a pesar de la carga del sistema
    - Puede significar una gran oscilación en la sobrecarga del sistema o la necesidad de poner a punto el sistema para eliminar las **inestabilidades**

4.4.2. Orientados al sistema

- Relacionados con el procesamiento
  - Rendimiento
    - Maximizar el número de procesos completados por unidad de tiempo
    - Medida de cuanto trabajo se está realizando
  - Utilización del procesador
    - Es el % de tiempo que el procesador está ocupado
    - Para un sistema compartido costoso, es un criterio significativo
- Otros
  - Equidad
    - Los procesos deben ser tratados de la misma manera, y ningún proceso debe sufrir inanición
  - Imposición de prioridades
    - Favorecer a los procesos con prioridades más altas
  - Equilibrado de recursos
    - Mantener ocupados los recursos del sistema
    - Favorecer los procesos que utilicen poco los recursos que en un determinado momento están sobreutilizados

4.5. Uso de prioridades



4.6. Algoritmos de planificación

4.6.1. Características

- Función de selección
  Basada en:
  - Prioridades
  - Requisitos sobre los recursos
  - Características de ejecución del proceso
    - **w**: tiempo usado en el sistema hasta ese momento
    - **e**: tiempo usado en ejecución hasta ese momento
    - **s**: tiempo total de servicio requerido por el proceso

- Modo de decisión
  Indica en que momento se aplicará la función
  - Sin expulsión (**no apropiativo**)
  - Con expulsión (**apropiativo**): Evitan monopolización
    - Cuando llega un nuevo proceso
    - Cuando llega una interrupción cambio de estado B → L
    - Periódicamente basado en interrupciones de reloj

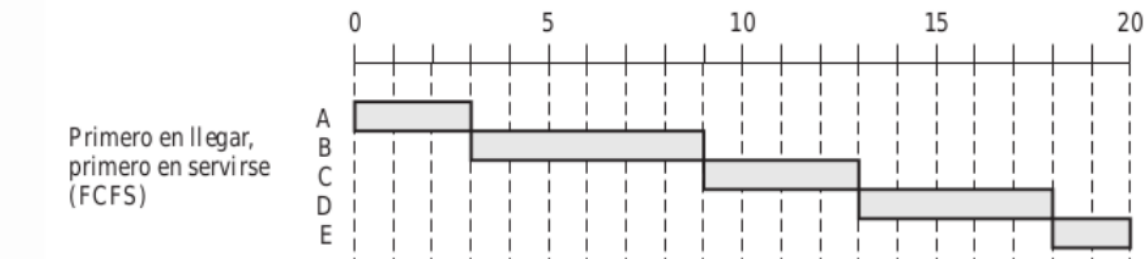
4.6.2. Algoritmos

- FCFS o **FIFO**
  - Siglas de *first-come-first-service* o *first-in-first-out*, lo que quiere decir que el primer proceso que entra es el primero en servirse
  - Cuando se deja de ejecutar el proceso actual, se selecciona para ejecutar el proceso que lleva **más tiempo en la cola de Listos**
  - No apropiativo
  - Tiende a favorecer **procesos limitados por el procesador** sobre los limitados por E/S
  - Perjudica a los procesos cortos (**no equitativo**)

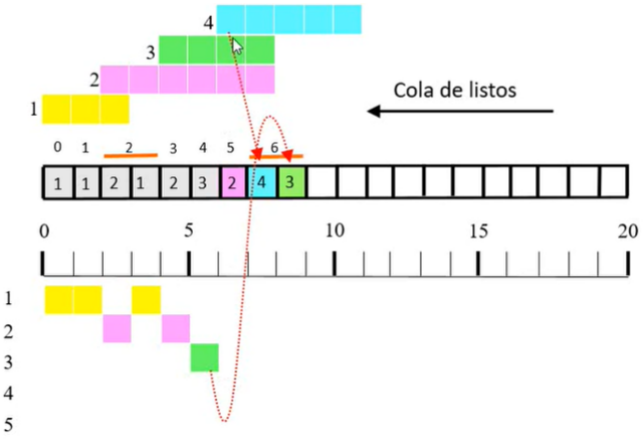
Proceso	A	B	C	D	E	
Tiempo de llegada	0	2	4	6	8	
Tiempo de servicio (T_s)	3	6	4	5	2	Media
Tiempo de finalización	3	9	13	18	20	
Tiempo de estancia (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56

$T_r = T_{finalización} - T_{llegada} = T_{espera} + T_s$

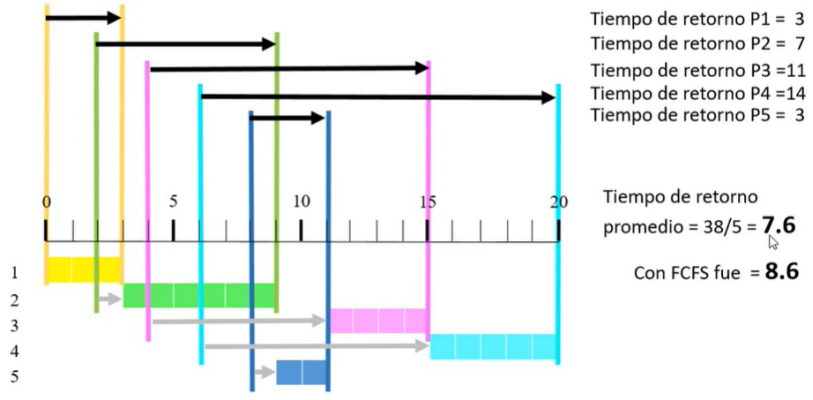
T_r normalizado = T_r / T_s (retardo relativo experimentado por un proceso)



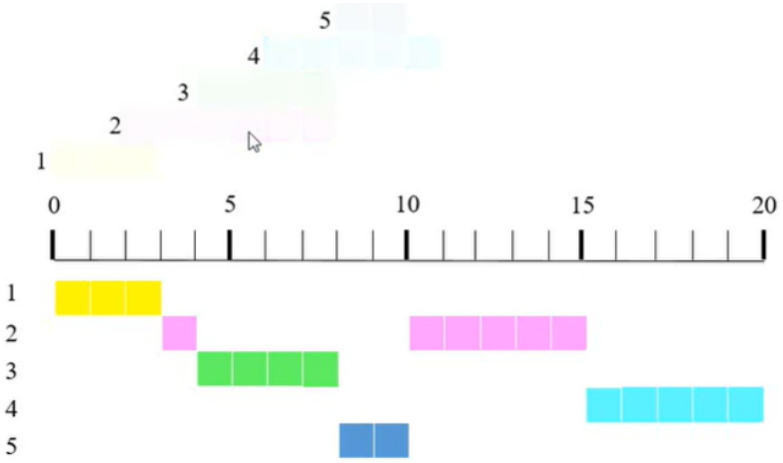
- **Round Robin**
  - Turno rotatorio o planificación cíclica
  - Reduce el castigo que tienen los procesos cortos con FIFO usando una expulsión basada en el reloj (FIFO lo hace con apropiación)
  - *time slicing*: A cada proceso se le da una rodaja o *quantum* de tiempo antes de ser expulsado (interrupciones de reloj por intervalos)
  - Efectivo en **sistemas de tiempo compartido** de propósito general o en **sistemas de procesamiento transaccional**
  - Mal rendimiento de los procesos limitados por E/S



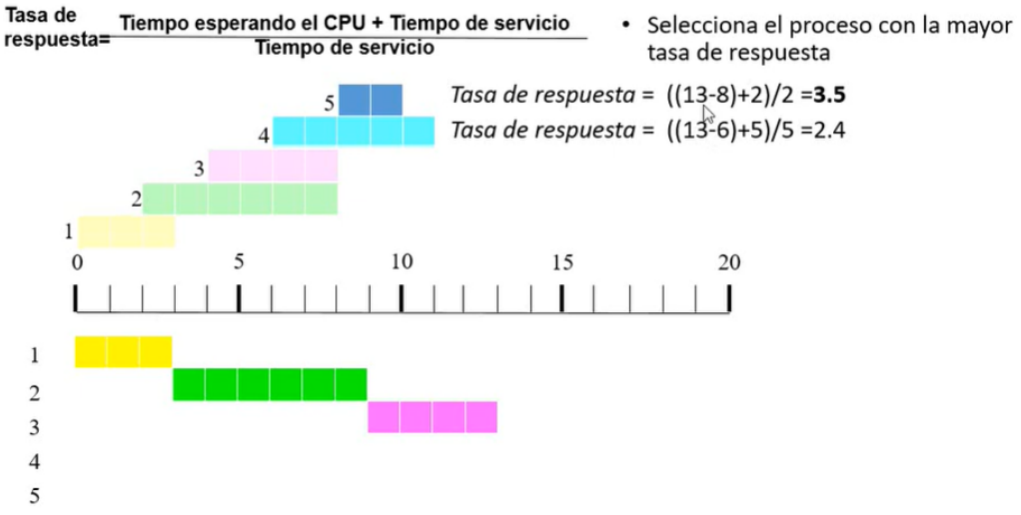
- Round Robin Virtual
  - Usa una **cola auxiliar** FIFO a la que se mueven los procesos después de estar bloqueados en una E/S
  - Los procesos de la cola auxiliar tienen prioridad sobre los de la cola de listos
  - Se ejecutan por un tiempo no superior al *quantum*, menos el tiempo total que ha estado ejecutando desde que no está en la cola de listos
- SPN o **SPF**
  - **Primero el proceso más corto** o *shortest process first*
  - No apropiativo
  - Mejora el rendimiento en términos de **tiempo de respuesta**
  - Se reduce predictibilidad (variabilidad tiempo de respuesta)
  - Requiere conocimiento o estimación del tiempo de procesamiento requerido por cada proceso
  - Riesgo de **inanición** para procesos largos



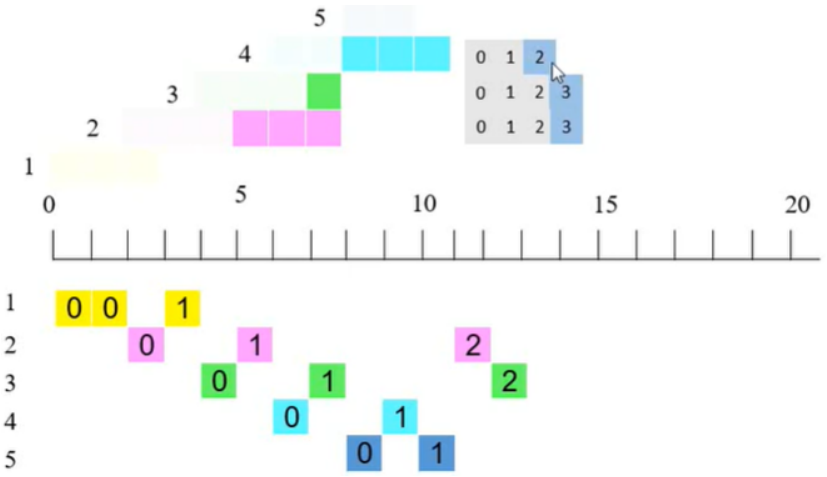
- **SRT**
  - **Menor tiempo restante** (*shortest remaining time*)
  - El planificador siempre escoge el **proceso que tiene menor tiempo restante** esperado
  - Versión **apropiativa** de SPN. El planificador puede expulsar el proceso actual si llega uno nuevo a la cola de listos con un tiempo restante menor
  - Requiere estimación del tiempo de procesamiento requerido por cada proceso
  - Riesgo de **inanición** para procesos largos



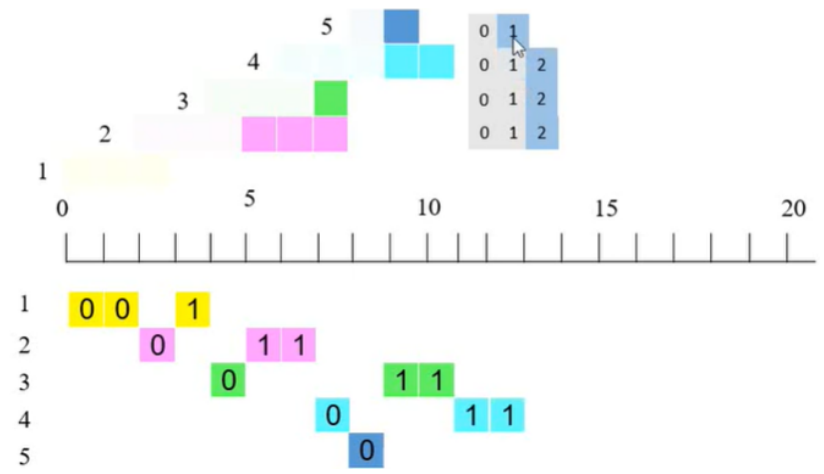
- **HRRN**
  - **Primero el de mayor tasa de respuesta** (*highest response ratio next*)
  - **R = (w+s) / s**, (**w**: tiempo esperando por CPU y **s**: tiempo de servicio esperado)
  - **No apropiativo**
  - Favorece a los **procesos cortos**, pero **evita la inanición de los largos** teniendo en cuenta el tiempo que llevan esperando
  - Requiere estimación del tiempo de procesamiento requerido por cada proceso



- Feedback
  - Si no se puede averiguar el T_s →dar preferencia a los procesos más cortos penalizando a los que han estado más tiempo en ejecución
  - **Apropiativo** por **cuantos de tiempo** y con mecanismo de **prioridades** dinámico
    - Cada vez que un proceso es expulsado se sitúa en la siguiente cola de menor prioridad
    - Dentro de cada cola se utiliza FIFO excepto en la de menor prioridad que se usa Round Robin
  - Favorece a los procesos nuevos más cortos frente a los más viejos y largos
  - Procesos largos → T_r excesivo, incluso **inanición**
  - Compensar variando tiempos de expulsión de cada cola (cuantos)
    - cola CL0 → q=1
    - cola CL1 → q=2
    - …
    - cola CLn → $q=2^n$
  - Solucionar inanición promoviendo a los procesos a colas de mayor prioridad tras esperar un tiempo en su cola actual
  - **Ejemplo para q = 1**



  - **Ejemplo para q = 2^n**



A continuación una tabla para **resumir los algoritmos** expuestos anteriormente de forma visual:

	Función de Selección	Modo de Decisión	Rendimiento	Tiempo de Respuesta	Rendimiento	Efecto sobre los Procesos	Inanición
FCFS	max[*w*]	No expulsiva	No especificado	Puede ser alto especialmente si hay mucha diferencia entre los tiempos de ejecución de los procesos	Mínima	Penaliza procesos cortos; penaliza procesos con mucha E/S	No
Turno Rotatorio *(round robin)*	constante	Expulsiva (por rodajas de tiempo)	Puede ser mucho si la rodaja es demasiado pequeña	Proporciona buen tiempo de respuesta para procesos cortos	Mínima	Tratamiento justo	No
SPN	min[*s*]	No expulsiva	Alto	Proporciona buen tiempo de respuesta para procesos cortos	Puede ser alta	Penaliza procesos largos	Posible
SRT	min[*s-e*]	Expulsiva (a la llegada)	Alto	Proporciona buen tiempo de respuesta	Puede ser alta	Penaliza procesos largos	Posible
HRRN	max(*w+s/s*)	No expulsiva	Alto	Proporciona buen tiempo de respuesta	Puede ser alta	Buen equilibrio	No
Feedback	(ver texto)	Expulsiva (por rodajas de tiempo)	No especificado	No especificado	Puede ser alta	Puede favorecer procesos con mucha E/S	Posible

w = tiempo de espera
e = tiempo de ejecución hasta el momento
s = tiempo total de servicio requerido por el proceso, incluyendo *e*

Sobrecarga

A continuación una **comparación de sus tiempos**:

	Proceso Tiempo de llegada Tiempo de servicio (T_s)	A 0 3	B 2 6	C 4 4	D 6 5	E 8 2	Media
FCFS	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	3 3 1.00	9 7 1.17	13 9 2.25	18 12 2.40	20 12 6.00	 8.60 2.56
RR $q = 1$	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	4 4 1.33	18 16 2.67	17 13 3.25	20 14 2.80	15 7 3.50	 10.80 2.71
RR $q = 4$	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	3 3 1.00	17 15 2.5	11 7 1.75	20 14 2.80	19 11 5.50	 10.00 2.71
SPN	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	3 3 1.00	9 7 1.17	15 11 2.75	20 14 2.80	11 3 1.50	 7.60 1.84
SRT	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	3 3 1.00	15 13 2.17	8 4 1.00	20 14 2.80	10 2 1.00	 7.20 1.59
HRRN	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	3 3 1.00	9 7 1.17	13 9 2.25	20 14 2.80	15 7 3.5	 8.00 2.14
FB $q = 1$	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	4 4 1.33	20 18 3.00	16 12 3.00	19 13 2.60	11 3 1.5	 10.00 2.29
FB $q = 2^i$	Tiempo de finalización Tiempo de estancia (T_r) T_r/T_s	4 4 1.33	17 15 2.50	18 14 3.50	20 14 2.80	14 6 3.00	 10.60 2.63