TODO SISTEMAS OPERATIVOS

COSAS IMPORTANTES TEORÍA:

·SO → **Conjunto de procesos que administran los recursos** del **Hardware** y **Software** de una computadora.

·Proceso → Programa en memoria principal. Tipos:

-De sistema: Se encargan de **gestionar, controlar y facilitar el uso del SO**. Pensadas para administrar y manejar el sistema. (EJ: shells, comandos básicos, herramientas red, demonios, etc).

-De aplicación: Se encargan de **resolver problemas del usuario final**. (EJ: videojuegos, navegadores, hojas de cálculo, etc).

·Factores para gestión de recursos del sistema:

-Equitatividad: A todo proceso, tenga la prioridad o tipo que tenga, **se le debe dar acceso a un recurso** para así, **evitar la inanición** (fin tiempo limite en CPU)

-Respuesta diferencial: Sistema **debe responder de forma dinámica** dependiendo del tipo y prioridad del proceso..

-Eficiencia: **Maximizar productividad** (nº procesos acabados por tiempo) y **minimizar el tiempo de respuesta** (tiempo hasta que un proceso es atendido).

·Planificador a corto plazo → Es el **encargado de decidir que procesos pasan de estar de estado "Listo"** (cola corto plazo**) a estado "Ejecutando"** (en CPU).

·Manejador de llamadas nativas al sistema → **Encargado de gestionar las llamadas nativas al sistema** (funciones a nivel de núcleo). Estas son accesibles a través de APIs (funciones a nivel de usuario).

·Manejador de interrupciones → **Encargado de gestionar las interrupciones** de E/S o del SO. Tipos:

-Interrupción por temporización: Cuando **un proceso agota el tiempo asignado en CPU**.

-Interrupciones por excepción**: Proceso sale de la CPU debido a una excepción** (división entre 0, *Segmentation Fault*, etc)

-Interrupciones de E/S: Cuando un dispositivo de E/S necesito hacer una operación**, proceso actual sale de CPU para antenderla**.

·Sistemas de Colas:

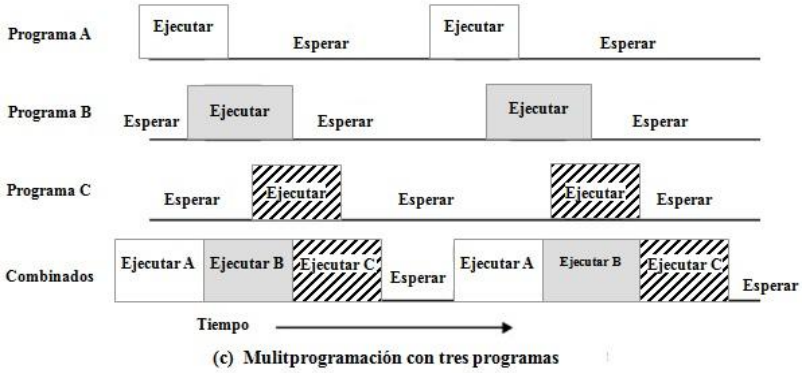
-Cola a corto plazo: **Procesos en RAM en estado "Listo"**.

-Cola a largo plazo: **Procesos esperando a entrar en CPU, pero que no están cargados completamente en RAM**. Necesitan pasar a la cola de corto plazo.

-Cola a corto plazo: **Procesos que quieren usar el mismo dispositivo de E/S**.

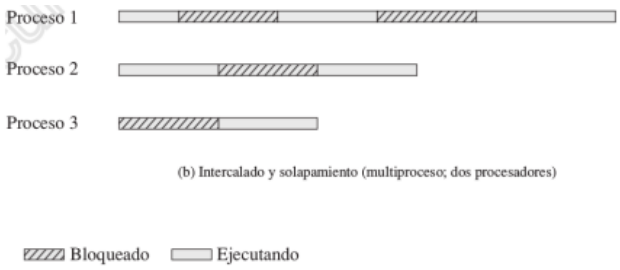
·Diseño modular del SO → Es la **separación del código del SO en componentes mas pequeñas** para **mayor libertad de cambio y ampliación**, para **menor esfuerzo de mantenimiento y** para **mayor facilidad de corrección de errores**.

·Multiprogramación → Es la **capacidad de intercambio de procesos de la CPU a lo largo del tiempo**. Esto permite que el SO meta en CPU procesos diferentes cuando haya una interrupción.



(c) Mulitprogramación con tres programas

·Multiprocesamiento → Son **CPU con varios procesadores, por lo que pueden hacer varios procesos simultáneamente**. TODOS los núcleos pueden realizar las mismas funciones.



(b) Intercalado y solapamiento (multiproceso; dos procesadores)

·Microprocesadores de propósito especial → Son procesadores pertenecientes a otros componentes/periféricos del sistema (Ej: tarjeta gráfica, etc). Estos procesadores no hacen que el sistema sea multiprocesador.

·Compilador → **Programa encargado de pasar código,** en X lenguaje (en C, C++, GO, etc) **a código maquina** (binario).

·CPU → **Chip que controla el funcionamiento del computador** mediante la ejecución de los programas de sistema. Ejecuta instrucciones alojadas en RAM. Se compone de:

-ALU: Es la **unidad encargada de hacer operaciones matemáticas** y lógicas.

-Registros intercambio de datos:

1) RDIM / MAR: **Almacena dirección de memoria de RAM donde leer o escribir**.
2) RDAM / MDR: **Almacena datos que se reciben de RAM o queescribir en ella**.

-Registros ejecución de instrucciones:

1) IR: **Contiene la última instrucción leída y que hay que ejecutar** (indicada por PC).
2) PC: **Contiene la siguiente instrucción a cargar en RAM** (Valor actualizado después de cada paso de memoria a IR).
3) PSW: Registro con un **conjunto de bits de estado** (bit modo usuario/supervisor, bits de resultados, bit de overflow, etc).

-Registros auxiliares:

1) Ac: **Para el almacenamiento de resultados matemáticos de la ALU**.
2) Registros propósito general: Sirven para **almacenar temporalmente bits de estado o instrucciones** (AX, BX, etc).
3) Registros de la pila: **Registros encargados de apuntar a lo cima** (SP) **o a la base de la pila**.

·Puente Norte → **Componente que gestiona las comunicaciones con la CPU de alta velocidad** (Ej: RAM, tarjeta gráfica, etc). Hoy en día, integrado en la CPU.

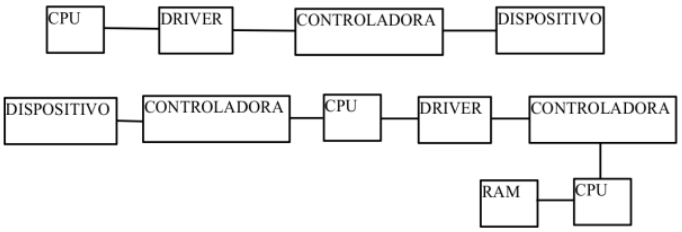
·Puente Sur → **Componente que gestiona las comunicaciones con la CPU de baja velocidad** (Ej: USB, discos duros, internet, etc). Hoy en día, sigue existiendo.

·Driver → **Componente software que permite al SO conectarse con el hardware y periféricos** y entender su funcionamiento.

·Operación E/S: **(* = a través de buses del sistema)**

1) Inicio de la llamada → **SO carga en CPU el driver** del dispositivo.
2) Driver → **Carga registros en el bus driver** sobre la instrucción*.

3) Bus Driver → **Analiza información** y **se comunica con el periférico***.
4) Periférico → **Realiza la acción**.
5) Periférico → **Manda resultado de la acción al bus driver***.
6) Bus driver → **Lee señal** y a su vez se la **envia a la CPU***.
7) SO → **Carga de nuevo en CPU el driver** del dispositivo.
8) Driver → **Traduce información** y **transfiere información a la RAM.**
9) SO → **Carga**, cuando considere oportuno, **el programa que espera esa respuesta en CPU**.

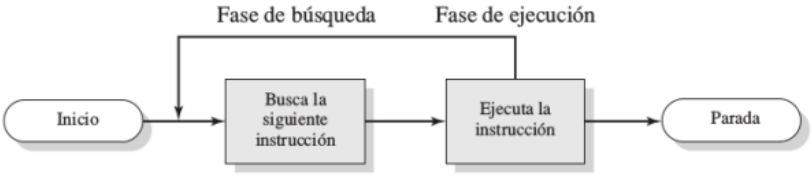


·***HyperThreading*** → Consiste en la **simulación de dos procesadores en uno solo para el procesamiento en paralelo de hilos.** Esto se consigue duplicando registros como el PC o el IR y dejar otros comunes, como la ALU.

·**Ciclo de instrucción** → **Procesamiento requerido para llevar a cabo la finalidad o ejecución de una instrucción**. Dos pasos:

-Fase de búsqueda: Consiste en **cargar y analizar la instrucción**. Se **carga en IR la instrucción indicada en PC**. Luego, **se interpreta el tipo de instrucción** cargada.

-Fase de ejecución: **Consiste en llevar a cabo la instrucción** ya cargada y analizada. **Se ejecuta la instrucción y se incrementa el valor de PC**, para continuar con el flujo del programa.



·**Interrupción síncrona** → : La interrupción **ocurre en un punto "predecible" durante la ejecución en CPU de la instrucción actual**, que es la que causa la interrupción.

·**Interrupción asíncrona** → **Evento que la causa es independiente de cualquier instrucción específica** que se esté ejecutando en ese momento en CPU.

·Interrupciones:

-Hardware: (asíncrona)

1) E/S: **Creada al entrar a un proceso de E/S** durante la ejecución de un programa.
2) Por fallo de hardware: Por acceso a zonas corruptas de memoria, dispositivo dañador, etc.

-Software: (síncrona)

1) Por temporización: Generadas cada cierto tiempo por el sistema **para tareas críticas** o cuando un **proceso agota su tiempo de estancia en CPU**.
2) Por llamadas al sistema: Ocurren cuando **se hace una llamada al sistema que a su vez hace una llamada nativa del sistema**. No es una interrupción como tal.

-Excepción: **Son condiciones de error generadas al ejecutar una instrucción**. (EJ: División por cero, Overflow, zonas de RAM corruptas, etc). Es una situación de error detectada por la CPU (hardware) en una instrucción (software). (síncronas)

·Controlador de interrupciones → Encarga de parte del **procesamiento de la gestión de interrupciones**, el resto es de la CPU (excepciones solo por la CPU). Esta hace:

-Recibe señales de interrupción: **Captura señales de interrupción** proveniente de los controladores de dispositivos.

-Ordena y prioriza señales.

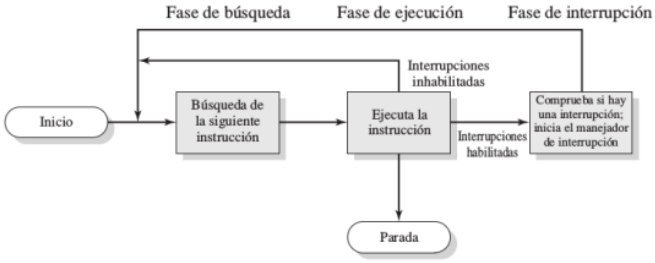
-Envia señal a la CPU: **Esta actualiza un registro de la PSW** y **envía la ID de la rutina ISR a cargar**.

·Rutinas ISR → Tipo de rutina encargada de **tratar una interrupción**.

·Manejador de interrupciones → Software del núcleo **encargado de consultar el tipo de interrupción recibida**. Luego, **transfiere la ejecución de la CPU a una posición de memoria fijada** para eso, la cual contiene la dirección de inicio de la rutina ISR a ejecutar. Cuando la **ISR acabe, o reanuda el proceso o reanuda otro diferente**.

·Vector de interrupciones → **tabla de punteros a las diferentes rutinas de interrupción ISR**. Manejador indica a CPU el índice de esta tabla para saber que rutina ISR ejecutar.

Adición de una **fase de interrupción** al **ciclo de instrucción** (figura 32).



·Código del SO → **Conjunto de instrucciones para gestionar recursos y proporcionar servicios**. Estas no son accesibles directamente para el usuario, ya que son funciones del kernel, solo **accesibles a través de llamadas nativas del sistema**. Por eso, existe un bit de estado (en PSW) que **cambia el modo de ejecución**.

·Modo núcleo (kernel) → **Asociado al procesamiento de instrucciones y rutinas del SO**. **Alojadas en parte de RAM reservada**, **no accesible** por el usuario. **Accesible mediante llamadas nativas al sistema**, previamente a través de APIs (llamadas al sistema).

·Modo usuario → **Asociado al procesamiento de instrucciones y rutinas del usuario**. Procesamiento de **rutinas cuyo código y datos estén alojados en la parte** de memoria de la **RAM reservada para usuarios**. Para **acceder a RAM del modo núcleo, usamos wrappers** (llamas al sistema).

·Pasos hasta llamada nativa desde APIs:

1) **Disponen parámetros asociados a la llamada al sistema** (en registros EBX, ECX, etc o en pila). **Guardar la ID de la llamada nativa en EAX**.
2) El **wrapper llama a una función de tipo TRAP** que ejecuta una instrucción especial **(INT 0X80 o SYSCALL)**, que es una **interrupción que cambiara el sistema a modo núcleo**.
3) Se **copian parámetros e ID de la llamada nativa a memoria del núcleo**
4) **Salvaguardado del "contexto"** actual del program, incluyendo PC.
5) **Se examinan ID y parámetros**. Se busca si esa ID existe y si los parámetros son correctos.
6) **Ejecución de la llamada nativa**. Machaca EAX con el valor de retorno y copiarlo en el proceso salvaguardado.
7) **Machaca EAX con el valor de retorno y copiarlo en el proceso salvaguardado**, además de **restaurar el contexto del proceso** y **volver a modo usuario** con funciones RETURN FROM TRAP.

·**Proceso**→ **Unidad de actividad cargada en memoria principal** caracterizada por un hilo principal y un estado actual, con sus datos y recursos del sistema asociados.

·**BCP** → **Contexto de un proceso**. Es el conjunto de datos por el cual el SO es capaz de supervisarlo y controlarlo. El BCP **permite la multiprogramación**, ya que permite que el proceso pare y se reanude. **Este se actualiza si cambia su estado** (Listo, Bloqueado, Zombie, etc), **cuando el proceso es interrumpido**, donde se guardan sus datos o **cuando cambia su tiempo en CPU**. Este struct contiene:

-ID: **ID único** de cada proceso. Almacenado también del user y el del padre del proceso.

-Estado: **Ejecutando, Listo, Bloqueado, Suspendido, Terminado, Zombie**.

-Información de planificación: **Prioridad** respecto a otros procesos.

-Límites de memoria asignados al proceso: Espacio de **direcciones asignadas al proceso en RAM.**

-Datos de contexto en relación a los registro de la CPU: PC, IR, bits de estado…

-Información de estado de E/S y recursos asignados

-Información de auditoria: **Sirve para el planificador y el funcionamiento interno del SO**. Estos son los tiempos en CPU y en espera.

·**Planificador a largo plazo** → **Encargado de cargar todo el código en RAM para su utilización**. Antes, solo cargada pequeña parte → BCP

·**Dispatcher** → **Parte del kernel del SO que se encarga de conmutar los procesos**. Dispatcher realizará el **salvado de proceso actual y la restauración del proceso** a conmutar. El planificador a corto plazo indicará cual debe sacar y cual meter.

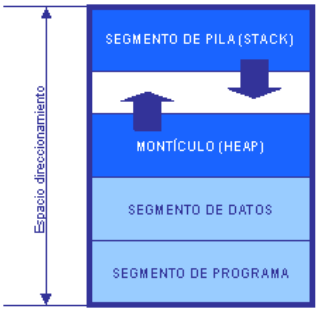
·**Partes de un proceso**:

-Zona de código: **Zona de memoria fija para almacenar el código fuente del proceso** en instrucciones máquina.

-Zona de datos: **Zona de memoria fija para almacenar las variables** globales, las constantes y las estáticas.

-Heap (Montículo): **Zona de memoria variable para la gestión dinámica de memoria** que se solicita durante la ejecución del proceso.

-Stack (Pila): Es una **parte del proceso en RAM encargada de controlar las llamadas a funciones, parámetros y retornos** de estos mismos (recursividad). Contiene un **conjunto ordenado de registros de activación** que almacenan información sobre las funciones invocadas. Es como una lista, de tipo FIFO (recursividad). Es de longitud variable durante la vida de esta.



·**SP** → **Puntero a la cima de la pila**. Esta apunta al ultimo elemento introducido en esta.

·**FP** → **Puntero a la base de la pila**. Esta apunta al primer elemento introducido en esta.

·**BIOS** → **Programa almacenado en un chip de memoria**. Las actuales son de EEPROM, lo que quiere decir que su firmware puede ser actualizado sin cambiar el chip. **Lo primero al arrancar una computadora**. Es ejecutada por la CPU y la más segura y actual es la UEFI.

·**Bootloader** → Al pasar las pruebas de la BIOS, esta manda a la **CPU a cargar al bootloader** en RAM (GRUB). Aquí ya se usan diferente tipos de esquemas de particiones como MBR o GPT.

Después, este se ejecuta en CPU. **Este código localiza el kernel del SO y lo carga en RAM**. Este entre en CPU y, durante este proceso, se hacen varias configuraciones (de memoria, de dispositivos, de archivos, de procesos, etc).

·**Kernel** → Al ser cargado, el SO comienza a establecer el espacio de usuario. Este carga varias configuraciones para hacer que el usuario sea capaz de usar la máquina.

·**Demonios** → **Son procesos en segundo plano encargados de tareas como** el bluetooth, la conexión a internet, el sonido, entorno gráfico. Se

ejecutan en entorno de usuario, accediendo al kernel mediante llamada al sistema.

·**Applet** → **Interfaz gráfica para conexión con un demonio**.

·**Finalización de un proceso:**

-Forma voluntaria:

1. Salida normal: **Termina su trabajo o lo terminamos manualmente** con, por ejemplo, un *ctrl+c* en la terminal.
2. Salida controlada por error: **Proceso termina debido a un error tomado en cuenta al crearlo.** Finaliza antes de llegar a este.

-Forma involuntaria:

1. Error fatal: **Terminación debido a un error no predecible**. Esto provoca una excepción.
2. Eliminación por otro proceso: **Proceso termina debido a otro proceso**, como cerrar la terminal o mandar un *kill()*.

·**Modelo de comportamiento para controlar la ejecución de los procesos:**

-Nuevo: **Proceso recién creado**. No esta completamente cargado en RAM, solo BCP. A la espera del planificador a largo plazo.

-Listo: **Proceso preparado para ser ejecutado**. Cargado completamente en RAM. A la espera del planificador a corto plazo.

-Ejecutando: **Proceso ejecutándose en CPU**.

-Bloqueado: **Proceso no puede ejecutarse en CPU hasta que un evento ocurra** (E/S, *wait()*, *pause()*, etc). Cuando ocurra, vuelve a estar en "Listo".

-Saliente: Proceso ha terminado. Ya no es elegible para ejecución y no esta cargado en RAM. Información temporalmente conservada (para ser recogida con *wait()*, por ejemplo).

·Hilo/Hebra → Es la unidad básica de utilización de la CPU con una serie de información asociada a dicho hilo. **Un proceso puede tener varios hilos, actuando en paralelo** si se lanzan a la vez (procesos son secuenciales). Cada hilo perteneciente a un proceso posee:

-Estado de ejecución: Visto anteriormente.

-BCP: Un **contexto asociado a ese hilo**.

-Registro de activación: **Dirección de retorno, parámetros, valor devuelto, variables locales, etc.**

Luego, el hilo comparte con el proceso al que pertenece y con los demás hilos:

-Zona de código: Estos se crean en el mismo código que el parámetro.

-Zona de datos estáticos: Las variables globales, por ejemplo. Cuidado, usar semáforos para evitar corrupción.

-Montículo: Las reservas de memorias en hilos, se mantendrán en el proceso padre, y viceversa.

·Niveles de ejecución:

-Alto (0): Apagado del sistema. Se ejecutan scripts para la finalización de demonios.

-Modo monousuario (1): **No configura la interfaz de red o los demonios** de inicio. **Solo un usuario, el root**. Permite reparas problemas o pruebas del sistema

-Modo multiusuario (2): **No configura la interfaz de red o los demonios** de inicio. **Permite varios usuario**.

-Multiusuario con red (3): **Inicia el sistema normalmente**. Carga de internet y demonios de inicio. **Solo interfaz en modo texto**.

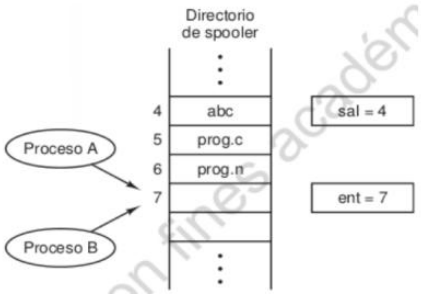
-Multiusuario con red (4): **Igual que el 3,** reservado para administradores de sistemas.

-Multiusuario gráfico (5): **Igual que el 3**, pero con **interfaz con entorno gráfico**. Default de arranque.

-Reinicio (6): **Reinicio del sistema**.

·**Procesos concurrentes** → Se dice esto si hay una **existencia simultánea de varios procesos en ejecución que actúan sobre recursos compartidos**.

·**Spooling:** → **Proceso mediante el cual la computadora introduce trabajos en un buffer, un área especial en memoria o incluso en un disco**. Esto causa problemas si dos procesos quieren introducir a la vez un objeto nuevo a ese buffer.



·**Sección crítica (SC):** → **Parte un programa donde se accede a recursos compartidos**. Necesitan exclusión mutua para funcionar correctamente.

·**Exclusión mutua** → Esto quiere decir que **nunca mas de un proceso debe estar ejecutando a la vez una SC**. Para ello, usamos semáforos. Cuando un **proceso entra a esta sección**, lo bloquea para que nadie entre, **y cuando sale**, lo desbloquea para que entre el siguiente.

·**Interbloqueo** → **Bloqueo permanente de un conjunto de procesos**. Esto ocurre cuando un **proceso esta bloqueado por otro proceso, y la parte que lo desbloquea esta bloqueda también**. Varios tipos de interbloqueo:

-Nivel de kernel: **Interbloqueo entre procesos que compiten por recursos** del sistema de uso exclusivo.

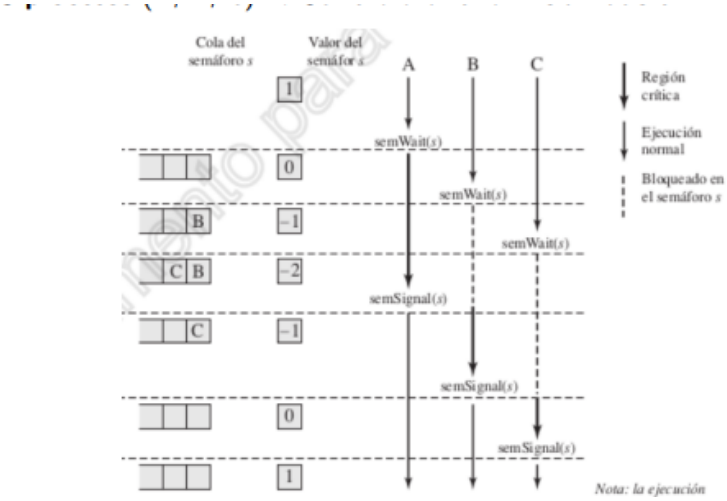
-Nivel de usuario: **Interbloqueo por mal uso de los mecanismos de exclusión mutua**.

P1	P2
(1) - Pide (escáner)	*(2) - Pide (impresora)*
(3) - Pide (impresora)	*(4) - Pide (escaner)*
Usa impresora y escaner	*Usa impresora y escaner*
Libera (impresora)	*Libera (escaner)*
Libera (escaner)	*Libera (impresora)*

·**Semáforo** → **Encargado de hacer que un proceso bloque una sección crítica y los demás no puedan entrar**. Dos tipos:

-Semáforo binario (mutex): Solo **permite la entrada de un hilo a esa SC**. El **mismo que lo bloqueo es el ÚNICO que podrá desbloquear** el semáforo. Estará inicializado a **0 si está cerrado** y **1 si está abierto**. La operación ***SemWaitB()* cambiará el valor a 0 si es 1**, y dejara al hilo **esperando si es 0**. ***SemSignalB()* cambiará a 1 si es 0**.

-Semáforo generales: **Permite la entrada de x hilos a esa SC** (x>0). Estos semáforos **pueden ser desbloqueados por hilos distintos a ellos mismos**. Estos **permiten sincronizar hilos** (productores-consumidores) o **permitir varios en la misma SC** (lectores-escritores). ***SemWait()* restará 1 al valor del semáforo**, si no es este negativo (bloquea) y ***SemSignal()* sumará 1 al valor de los semáforos** (desbloquea).



·**Tiempo de respuesta aceptable** → **Tiempo que transcurre desde que se crea o lanza un proceso** (estado Nuevo) **hasta que se obtienen sus primeros resultados**.

·**Degradación aceptable** → Ante una subida de carga, una **degradación aceptable es que el tiempo de respuesta no suba preocupantemente ni el rendimiento baje**.

·**Tiempo de estancia** → **Tiempo que transcurre desde el momento en que se crea un proceso,** estado Nuevo, **hasta el momento en que se completa,** estado Saliente. Tiempo de Estancia: Tiempo Finalización - Tiempo Llegada.

·**Tiempo de espera** → **Suma de tiempos que el proceso está esperando a ser servido**. Tiempo de Espera: Tiempo de Estancia – Tiempo de CPU o de servicio.

·**Tiempo de CPU** → **Tiempo** (o porcentaje) **que está ocupada la CPU por un proceso.**

·**Algoritmo de planificación** → Este es tipo de decisiones que tomara el planificador a corto plazo para que proceso meter en CPU. Tipos:

-Sin expulsión: **Planificador ejecuta proceso hasta que acabe de forma natural, tenga una operación de E/S o una excepción**.

-Con expulsión: El **planificador selecciona un proceso para ejecutarlo y la decisión de expulsarlo de la CPU se toma si sucede algún tipo de evento debido a la política de planificación** que haga que el proceso actual tenga que ser expulsado, **como una rodaja de tiempo (quantum) o que llegue a la cola de "Listos" un proceso de mayor prioridad**.

·Algoritmo FCFS (First Come First Served) → Este algoritmo es de tipo **no expulsivo**, donde los **procesos que van llegando se unen a una cola de listos, donde el primero que llega, es el siguiente en entrar en CPU**, cuando termine el proceso que está actualmente ahí. Tiene inconvenientes, como **que puede provocar inanición de procesos cortos si antes van procesos largos**.

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

Figura 2: Tabla de información de los procesos, política FCFS.

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

·Algoritmo de turno rotatorio → **Expulsivo**. **Permite corregir el inconveniente del algoritmo anterior, añadiendo una rodaja de tiempo, o quantum, en la cual podrán estar en CPU. Cuando termine ese tiempo, vuelven a la cola de listos**. Si a la vez llegan un proceso que acaba de salir de CPU y uno nuevo, el **nuevo irá antes**.

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3				
B	2	6				
C	4	4				
D	6	5				
E	8	2				

Figura 5: Tabla de información de los procesos, política Round Robind. q=4

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

·Algoritmo de turno rotatorio virtual → **Expulsivo**. Igual que el rotativo normal, **pero los procesos "Bloqueados" por E/S**, cuando vuelven a "Listos", **entran en una cola de Listos auxiliar que tendra prioridad frente a la normal**, de procesos nuevos. Este **proceso entrará con la rodaja de tiempo sobrante de su entrada anterior, o con una nueva si ya habia terminado**.

Turno Rotatorio Virtual (***Virtual Round Robind*** o **VRR**) con ***q=4***. En el instante o **ciclo 6**, el proceso **B** realiza una operación de **E/S** que dura hasta el **7** (1 ciclo, el 6-7), y vuelve a estar **disponible** para ejecución en el instante o **ciclo 9**. Por otro lado, en el instante o **ciclo 14**, el proceso **D** realiza una operación de **E/S** que dura hasta el **15** (1 ciclo, el 14-15), y vuelve a estar **disponible** para ejecución en el instante o ciclo **17**.

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3	0	3	3	0
B	2	6	3	13	11	5
C	4	4	7	11	7	3
D	6	5	13	20	14	9
E	8	2	15	17	9	

Figura 18: Tabla de información de los procesos, política Round Robind Virtual.

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

·Algoritmo de planificación mediante colas de prioridad multinivel → **Expulsivo** (rodaja de tiempo) . Aquí **se le asigna a cada proceso una prioridad de las x que hay. Cuando un proceso salga de la CPU y no haya ninguno, se meterá el siguiente de la CL0. Si no hay, se examinará la siguiente, así sucesivamente**. Si proceso bloqueado vuelve de listos, cumpliría de nuevo rodaja de tiempo completa.

Colas de prioridades multinivel ***q=1*** (se **sobreentiende** que hay **3 colas**, **una por prioridad**). Un valor de **1** da **mayor prioridad** que un valor de **2**, y así sucesivamente.

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

Figura 25: Tabla de información de los procesos, política prioridades multinivel.

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

·Algoritmo de planificación mediante colas de prioridad multinivel retroalimentada→ **Expulsivo** (rodaja de tiempo) . Aquí **se le asigna a cada proceso una prioridad de las x que hay. Cuando un proceso salga de la CPU y no haya ninguno, se meterá el siguiente de la CL0. Si no hay, se examinará la siguiente, así sucesivamente**. Si un **proceso sale del procesador y estaba en la cola CLi, pasará a su consecutiva**. Si proceso bloqueado vuelve de listos, cumpliría de nuevo rodaja de tiempo completa.

Políticas de prioridades multinivel retroalimentada $q=2^i$ con **2 colas**, siendo **i=0** la **cola inicial con mayor prioridad**.

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

Figura 32: Tabla de información de los procesos, política prioridades multinivel retroalimentada $q=2^i$.

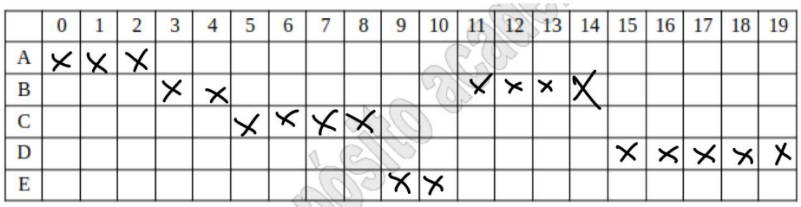
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	X	X	X																	
B				X		X	X					X	X					X		
C					X					X	X						X			
D								X						X	X				X	X
E									X							X				

·Algoritmo de planificación SPN (primero el más corto) → **No expulsivo**. Se **escoge primero al proceso que tenga menos ciclos de reloj**. **Recalculo de la cola de listos** se hace cuando proceso en CPU acabe. Si tienen mismo tiempo de servicio, se prioriza al que mas tiempo lleve (FIFO).

Política SPN. El proceso **B** hace una operación de **E/S** en el **ciclo** de reloj **4** (se entiende que la finaliza en el 5). **B** vuelve a estado **Listo** en el **ciclo 9**.

Proceso	Llegada	Tº CPU	Tº Inicio	Tº Fin	Tº Estancia	Tº Espera
A	0	3				
B	2	6				
C	4	4				
D	6	5				
E	8	2				

Figura 42: Tabla de información de los procesos, política SPN.



·Algoritmo de planificación SRT (primero menor tiempo restante) → **Expulsivo** (sin quantum). Entra en **CPU el proceso al que le queden menos ciclos de reloj**. **Si entra un nuevo proceso a la cola de listos con menos tiempo que el proceso en CPU, se meterá al nuevo**.

SRT con procesos limitados por el procesador.
Fíjese que en el **instante 10**, tanto el proceso **B** como **D** tienen el **mismo tiempo restante**, pero como **B entró** en la lista de **Listos** antes que **D**, se le da la **prioridad** a **B**.

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

Figura 46: Tabla de información de los procesos, política SRT.

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

COSAS IMPORTANTES PRÁCTICAS:

·PROCESOS (P1):

-Creacion de un proceso:

pid_t fork (void)

-Identificación de procesos:

pid_t getpid (void) / pid_t getppid (void) (id del padre)

-Suspensión y espera de un proceso: *pid_t wait (int *status)*

-Ejecutar proceso:

1) execl (execl, execlp) **pasan los argumentos de la línea de comandos como una lista, y es útil si se conoce el número de argumentos en tiempo de compilación**:

   *int execl(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/)*

   *int execlp(const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/)*

2) execv (execv, execvp) **pasan los argumentos de la línea de comandos en un array de argumentos, y es útil cuando no se conoce el número de argumentos en tiempo de compilación**:

   *int execv(const char *path, char *const argv[])*

   *int execvp(const char *file, char *const argv[]);*

```c
int main(int argc, char* argv[]){

    if(argc<3){perror("pocos argumento"); exit(-1);}

    int status;
    pid_t pid, childpid;

    for (int i=1; i<=2; i++){
        pid=fork();

        switch (pid){
        case -1:
            perror("fork error");
            printf ("error: %s", strerror(errno));
            exit(EXIT_FAILURE);

        case 0:
            printf("Soy el hijo %d, con ID %d y padre con ID %d \n",i, getpid(), getppid());

            if(i==1){
                if (execlp("gnome-calculator", "gnome-calculator", NULL)<0){
                    perror("Error al ejecutar la calculadora");
                    exit(EXIT_FAILURE);
                }
            } else if(i==2){
                if (execvp(argv[1], &argv[1])<0){
                    perror("Error al ejecutar gedit");
                    exit(EXIT_FAILURE);
                }
            }

            exit(EXIT_SUCCESS);
        }
    }

    while ((childpid = wait(&status)) > 0){
        if (WIFEXITED(status)){
            printf("Soy el padre %d, hijo %d recogido, status=%d\n", getpid(), childpid, WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)){
            printf("Proceso padre %d, hijo con PID %d finalizado al recibir la señal %d\n", getpid(), childpid, WTERMSIG(status));
        }
    }

    // Manejo de errores de wait fuera del bucle.
    if (childpid == -1 && errno == ECHILD){
        printf("Proceso padre %d, no hay más hijos que esperar. Valor de errno = %d, definido como: %s\n", getpid(), errno, strerror(errno));
    } else {
        perror("Error en la invocación de wait o waitpid");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

-Capturar señales:

void signal (int sig, void (*func)(int))* (cuando detecta esa señal, llama a la función)

-Enviar señales:

void kill(pid_t pid, int sig)

-Alarma:

unsigned int alarm(unsigned seconds) (cuando termina tiempo, manda señal)

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

void sig_alrm(int signo)
{
    printf("RING!!!\n");
    return;
}

void f1(int nsecs)
{
    alarm(nsecs);
    pause();
}


int main(){
    int i=0;

    if(signal(SIGALRM, sig_alrm)== SIG_ERR){
        perror("Signal error");
        printf("errno value= %d\n", errno);
        exit(EXIT_FAILURE);
    }

    printf ("Una alarma en 5 segundos...\n");
    f1(5);
    printf ("Una alarma en 3 segundos...\n");
    f1(2);
    while(1){
        printf ("Una alarma en 1 segundos...\n");
        i++;
        f1(1);
        if(i==4){kill(getpid(),9);}
    }
}
```

·HILOS (P2):

-Creación hebras:

*int pthread_create(pthread_t * thread, NULL, void * (*func) (void *), void *arg)*

-Esperar finalización hebras:

*int pthread_join (pthread_t th, void ** return)*

-Finalizar hebras:

*void pthread_exit (void *retval)*

-Obtener información hebras:

pthread_t pthread_self(void)

```c
void *rand_num(){
    float x = drand48()*10.1;
    printf("    |HEBRA HIJA| : Numero x=%.3f \n", x);
    float y = drand48()*10.1;
    printf("    |HEBRA HIJA| : Numero y=%.3f \n", y);

    double * z = malloc(sizeof(double));
    *z=x+y;
    printf("    |HEBRA HIJA| : Suma=%.3f\n", *z);

    pthread_exit((void*) z);
}

int main(int argc, char **argv){
    int num_threads = atoi(argv[1]);
    pthread_t threads[num_threads];
    int error, valor_join;
    double *ret;
    double valor=0;

    srand48 (time(NULL));
    for(int t=0;t<num_threads;t++)
    {
        printf("|HEBRA PADRE| : Creando hebra %d...\n", t);
        error = pthread_create(&threads[t], NULL, (void *) rand_num, NULL);

        if (error)
        {
            perror("Error joining thread\n");
            printf("errno value= %d\n", errno);
            exit(EXIT_FAILURE);
        }
    }

    for(int i=0;i<num_threads;i++)
    {
        valor_join = pthread_join(threads[i], (void**)&ret);

        if(valor_join!=0)
        {
          perror("Fallo en pthread_join()...\n");
          exit(EXIT_FAILURE);
        }

        valor+=*ret;
    }

    printf("|HEBRA PADRE| : Suma total=%.3f...\n", valor);

}
```

·SEMÁFOROS (P3):

-Creación mutex:

int pthread_mutex_init(pthread_mutex_t *mutex, NULL)

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

-Bloqueo mutex:

*int pthread_mutex_lock (pthread_mutex_t *mutex)*

-Desbloqueo mutex:

*int pthread_mutex_unlock (pthread_mutex_t *mutex)*

-Destrucción mutex:

*int pthread_mutex_destroy(pthread_mutex_t *mutex)*

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int *camisetas;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *cliente(void *arg) {
  int s;
  int num = rand() % 10 + 1;
  int numC = *((int *)arg);
  int tipo_cami = rand() % numC;

  // SECCIÓN CRITICA
  s = pthread_mutex_lock(&mtx);
  if (s != 0) {
    printf("Mutex lock error 1...\n");
    pthread_exit(NULL);
  }

  if (num < camisetas[tipo_cami]) {
    camisetas[tipo_cami] -= num;
    printf("    ||Cliente|| Un cliente ha comprado %d camisetas del tipo %d . "
           "Restantes: %d\n\n",
           num, tipo_cami + 1, camisetas[tipo_cami]);
  } else {
    camisetas[tipo_cami] = 0;
    printf("    ||Cliente|| Un cliente ha comprado todas las existencias del "
           "tipo %d \n\n",
           tipo_cami + 1);
  }

  s = pthread_mutex_unlock(&mtx);
  if (s != 0) {
    printf("Mutex_unlock error 1...\n");
    pthread_exit(NULL);
  }
  // SECCIÓN CRITICA

  pthread_exit(NULL);
}
```

-Creación semáforo:

*int sem_init(sem_t *sem, int pshared, int value)* (pshared en 0 si es para hilos, 1 para procesos)

-Bloqueo semáforo:

*int sem_wait(sem_t *sem)*

-Desbloqueo semáforo:

*int sem_post(sem_t *sem)*

-Destrucción semáforo:

*int sem_destroy(sem_t *sem);*

```c
  error = sem_init(&pr, 0, TAM_BUFFER);
  if (error != 0) {
    printf("sem_init pr error...\n");
  }
```

```c
sem_t cn;
sem_t pr;
int buffer[TAM_BUFFER];

void *productor(void *arg) {
  int n = *((int *)arg);

  for (int i = 0; i < n; i++) {
    sem_wait(&pr);
    pthread_mutex_lock(&s);
    buffer[i % TAM_BUFFER] = rand() % 11;
    printf("Colocado en la posicion %d el valor %d \n", i % TAM_BUFFER,buffer[i % TAM_BUFFER]);
    pthread_mutex_unlock(&s);
    sem_post(&cn);
  }
}
```

RECURSOS PARA MÁS:

·Open Group → https://pubs.opengroup.org/onlinepubs/9799919799.2024edition/

·Biblioteca glib → https://sourceware.org/glibc/manual/

·Moodle → https://moodle.uco.es/m2526/