* 1)
  + **Programa:** Es un conjunto de instrucciones escritas en un lenguaje de programación, que indican al computador qué tareas realizar. Es un archivo estático que no está en ejecución, y por lo tanto, no requiere recursos de sistema (como CPU, memoria, etc.) mientras permanece en esta forma. No tiene program counter y existe desde que se edita hasta que se borra.

**Proceso:** Es una instancia en ejecución de un programa. Cuando un programa se ejecuta, se convierte en un proceso y, en ese momento, se le asignan recursos del sistema (como tiempo de CPU, memoria, y acceso a dispositivos de E/S). Un proceso es dinámico y puede crear otros procesos, tener estados (como listo, en ejecución, bloqueado, etc), tiene program counter, es gestionado por el sistema operativo para realizar tareas específicas y su ciclo de vida comprende desde que se lo ejecuta hasta que termina.

* + **Tiempo de Retorno (TR):** Es el tiempo total que transcurre desde que un trabajo (job) llega al sistema hasta que se completa su ejecución. Incluye el tiempo de espera en la cola, el tiempo en ejecución, y cualquier otro tiempo necesario para que el trabajo termine. En otras palabras, se mide desde el instante de llegada hasta la finalización.

**Tiempo de Espera (TE):** Es el tiempo total que un trabajo pasa esperando en la cola de procesos listos antes de que se le asigne la CPU. No incluye el tiempo en ejecución ni el tiempo de espera por otros recursos; únicamente mide el tiempo de espera en la cola, hasta que el proceso comienza a ejecutarse.

* + **Tiempo Promedio de Retorno (TPR):** Es el promedio de los tiempos de retorno de todos los trabajos en un lote. Representa el tiempo promedio que toma completar cada trabajo desde su llegada al sistema hasta su finalización. Se calcula sumando los tiempos de retorno individuales de cada trabajo y dividiendo por el número total de trabajos.

**Tiempo Promedio de Espera (TPE):** Es el promedio de los tiempos de espera de todos los trabajos en el lote. Indica el tiempo promedio que cada trabajo pasa esperando en la cola antes de recibir atención (sin incluir el tiempo de ejecución). Se calcula sumando los tiempos de espera individuales de cada trabajo y dividiendo por el número total de trabajos.

* + **Quantum** es el intervalo de tiempo fijo que se asigna a cada proceso. Durante el tiempo del quantum, el proceso tiene el control exclusivo de la CPU. Si el proceso no finaliza en ese lapso, es interrumpido y enviado de vuelta a la cola de procesos listos, y el siguiente proceso en la cola recibe su turno.

**Quantum muy corto:** Puede causar un alto número de cambios de contexto, lo que aumenta la sobrecarga y reduce la eficiencia del sistema.

**Quantum muy largo:** Se acerca a un enfoque de planificación FCFS (First-Come, First-Served), reduciendo el beneficio de la equidad en la atención de procesos y la capacidad de respuesta en sistemas interactivos.

* + **Short Term Scheduler (Planificador a Corto Plazo):** También conocido como el planificador de CPU, decide cuál de los procesos en la cola de procesos listos debe ser el siguiente en obtener acceso a la CPU. Realiza esta tarea con alta frecuencia, ya que los procesos cambian de estado constantemente. Su objetivo es maximizar la utilización de la CPU y minimizar los tiempos de espera.

**Long Term Scheduler (Planificador a Largo Plazo):** Controla la admisión de nuevos procesos en el sistema, decidiendo cuántos y cuáles procesos deben ser cargados en la memoria principal y añadidos a la cola de procesos listos. Este planificador actúa con menor frecuencia y está más enfocado en controlar el nivel de multiprogramación, es decir, el número de procesos que residen en la memoria principal. El objetivo es equilibrar la carga de trabajo y optimizar el rendimiento global del sistema.

**Medium Term Scheduler (Planificador a Mediano Plazo):** Gestiona el intercambio de procesos entre la memoria principal y la memoria secundaria (swap), una técnica conocida como *swapping*. Su función es decidir qué procesos pueden ser temporalmente suspendidos y almacenados en disco para liberar memoria principal cuando el sistema está muy cargado. Este scheduler ayuda a mantener un uso eficiente de la memoria y a evitar que el sistema se sobrecargue.

* + El **Dispatcher** (o despachador) es un componente clave del sistema operativo que se encarga de la transición de control de la CPU entre procesos, ejecutando las siguientes tareas:
* **Cambio de Contexto:** Guarda el estado del proceso actual (registros, contador de programa, etc.) y carga el estado del proceso siguiente en la CPU. Este cambio de contexto permite que el sistema pueda retomar la ejecución del proceso suspendido en el mismo punto en que fue interrumpido.
* **Cambio de Modo de Usuario/Kernel:** Si es necesario, el dispatcher cambia la CPU del modo de kernel (donde se ejecutan operaciones de bajo nivel y acceso a hardware) al modo de usuario, donde los procesos de usuario pueden ejecutar sus instrucciones.
* **Salto al Lugar Apropiado en el Programa:** Establece el contador de programa (program counter) en la dirección correcta del proceso al que se le va a asignar la CPU, de manera que pueda continuar su ejecución en el punto exacto donde fue detenido o comenzar desde el inicio si es un proceso nuevo.
* 2)
  + El comando top permite ver los procesos que están corriendo en el sistema en tiempo real y proporciona una visión general del uso de recursos del sistema. Es especialmente útil para administrar el sistema y monitorear el rendimiento.
  + htop es una herramienta de monitoreo interactiva que muestra información detallada sobre los procesos y el uso de recursos en un sistema Unix/Linux. Es muy popular debido a su interfaz visual mejorada, su facilidad de uso, y su capacidad para gestionar procesos de manera más intuitiva que top. Instalación: sudo apt-get install htop
  + El comando ps (process status) muestra una instantánea del estado de los procesos en ejecución en el sistema en un momento dado. A diferencia de top y htop, que muestran procesos en tiempo real, ps simplemente lista los procesos activos en el momento en que se ejecuta el comando. ps proporciona información útil sobre los procesos, como el PID (ID del proceso), el usuario que ejecuta el proceso, el uso de recursos, el tiempo de ejecución y otros detalles que ayudan a comprender el estado actual de los procesos en el sistema.
* **ps aux:** Muestra todos los procesos en ejecución en el sistema, incluidos aquellos de otros usuarios y los procesos sin un terminal asociado.
* **ps -ef:** Muestra una lista de todos los procesos en un formato de árbol, con información como el ID del proceso padre (PPID).
* **ps -u [usuario]:** Muestra los procesos específicos que están siendo ejecutados por un usuario determinado.
  + pstree permite ver la jerarquía de procesos en una estructura de árbol, mostrando de forma gráfica cuál es el proceso raíz y qué procesos han sido creados a partir de otros.
  + El comando kill en Unix/Linux se utiliza para **enviar señales a procesos** en ejecución, siendo la más común la señal para finalizar o "matar" un proceso.
* **SIGTERM (señal 15):** Es la señal de terminación estándar. (kill PID)
* **SIGKILL (señal 9):** Es una señal forzada que finaliza el proceso de inmediato, sin darle tiempo de realizar limpieza. (kill -9 PID)
* **SIGHUP (señal 1):** Envía una señal de "hang up" (cierre de terminal). Esto puede hacer que algunos procesos vuelvan a cargar su configuración sin reiniciar completamente. (kill -1 PID)
* **SIGSTOP y SIGCONT:** SIGSTOP pausa un proceso sin finalizarlo, mientras que SIGCONT reanuda un proceso pausado. (kill -STOP PID)(kill -CONT PID)
  + pgrep se usa para **buscar procesos por su nombre** o ciertos criterios y devolver sus PIDs (ID de procesos) en la salida.

pkill es similar a pgrep, pero en lugar de solo buscar procesos, **también les envía una señal directamente**. Este comando es muy útil para finalizar procesos sin necesidad de conocer sus PIDs.

* + El comando killall en Unix/Linux se utiliza para **finalizar todos los procesos que coincidan con un nombre específico**. A diferencia del comando kill, que requiere el PID (ID de proceso) específico, killall permite terminar procesos simplemente proporcionando el nombre del proceso.
  + El comando renice ajusta el **valor de niceness** de uno o varios procesos, lo que afecta su prioridad. El **niceness** es un valor numérico que oscila entre -20 (máxima prioridad) y 19 (mínima prioridad). El valor predeterminado para los procesos es 0.
  + xkill es útil cuando una ventana de una aplicación no responde o no se puede cerrar de manera normal, permitiendo al usuario finalizar la aplicación sin necesidad de utilizar el administrador de tareas o comandos más complejos.
  + El comando atop en sistemas Unix/Linux es una herramienta avanzada de **monitoreo de rendimiento** en tiempo real. Se utiliza para **ver el uso de recursos del sistema** (CPU, memoria, disco, red, procesos, etc.) y proporciona información más detallada que otras herramientas como top o htop. atop está diseñado para proporcionar un análisis más completo y persistente de la actividad del sistema, y puede ser utilizado para **monitorear y almacenar registros históricos**.
  + Este bucle se ejecuta **3 veces**. Dentro de cada iteración, se llama a la función fork(), la cual crea un **nuevo proceso hijo**. fork() retorna dos valores:
* **En el proceso padre**: Retorna el PID (identificador de proceso) del hijo recién creado.
* **En el proceso hijo**: Retorna 0.

Es importante notar que en cada iteración del bucle se crean **más procesos**. Después de la primera iteración, el proceso principal (padre) crea un hijo. En la segunda iteración, tanto el proceso padre como el primer hijo creado llaman a fork(), lo que crea **otros dos procesos**. En la tercera iteración, cada uno de los cuatro procesos creados en las iteraciones anteriores vuelve a llamar a fork(), lo que genera **más procesos hijos (4)**.

El número exacto de veces que se imprimirá "Proceso" depende de cuántos procesos se hayan creado en total. Como hay un total de 8 procesos al final (1 proceso original + 7 procesos hijos creados por las llamadas fork()), se imprimirá "Proceso" **8 veces**.

Sí, el número de líneas que imprimen la palabra “Proceso” son la cantidad de procesos que se crean.

* + El ciclo for ejecuta fork() tres veces, lo que crea **8 procesos** (1 original + 7 hijos). Cada vez que se llama a fork(), se crea un nuevo proceso. Esto da lugar a una multiplicación exponencial de los procesos. Después de 3 iteraciones de fork(), tendrás 8 procesos en total.

La variable p está inicialmente en 0. Después de cada llamada a fork(), cada proceso tiene su propia copia de la variable p.

Después de que todos los procesos hayan creado sus copias de p, cada uno incrementa p de 0 a 1 (es decir, p++), y luego imprime el valor de p.

Se imprimirá “Proceso 1” por cada proceso.

Si colocamos el incremento de la variable dentro del bucle for, se mostrarán 8 líneas con el valor “Proceso 3”.

* + Una **pipe** es un mecanismo de comunicación unidireccional entre dos procesos. Es un canal de comunicación que permite que un proceso (el productor) envíe datos a otro proceso (el consumidor), que los recibe de forma secuencial.

Permiten:

* **Procesamiento en tiempo real:** Permite el procesamiento en tiempo real de la salida de un comando antes de ser utilizada por otro.
* **Combinación de comandos:** Facilita la comunicación entre comandos para lograr tareas más complejas.
  + Para crear un **pipe** en C, se utiliza la función pipe(), que es una llamada al sistema proporcionada por Unix/Linux. Esta función permite crear un canal de comunicación unidireccional entre dos procesos. Es necesario manejar un arreglo de enteros de dos posiciones para almacenar los descriptores de archivo (números que el sistema operativo usa para identificar y manipular archivos y canales de comunicación como los pipes) donde en la primera se usará para hacer referencia al extremo de lectura del pipe y la segunda posición hará referencia al extremo de escritura. “pipe(fd[N]) siendo N 0 ó 1 y fd un arreglo de enteros”.
  + El parámetro necesario para la creación de un **pipe** en C es un **array de enteros** que se pasa como argumento a la función pipe().
* **pipefd[2]**: Este es un arreglo de dos enteros que se utiliza para almacenar los descriptores de archivo que representan los extremos del pipe.
* **pipefd[0]**: Este es el descriptor de archivo del **extremo de lectura** del pipe. Los datos escritos en el extremo de escritura pueden ser leídos desde este extremo.
* **pipefd[1]**: Este es el descriptor de archivo del **extremo de escritura** del pipe. Los datos que se escriben en este extremo serán leídos desde el extremo de lectura.
  + **Unidireccional:** (Un solo flujo de datos) Un pipe permite que los datos fluyan en una dirección. Es decir, un proceso escribe en el pipe y otro proceso lee de él. No es posible que dos procesos escriban o lean desde un mismo pipe al mismo tiempo.

**Comunicación entre procesos (IPC)**: Las pipes permiten la comunicación entre dos procesos, donde uno actúa como **emisor (escritor)** y el otro como **receptor (lector)**. Los procesos pueden estar en el mismo programa (como en el caso de un **padre e hijo**), o incluso pueden ser procesos diferentes, siempre y cuando se pasen la información a través del pipe.

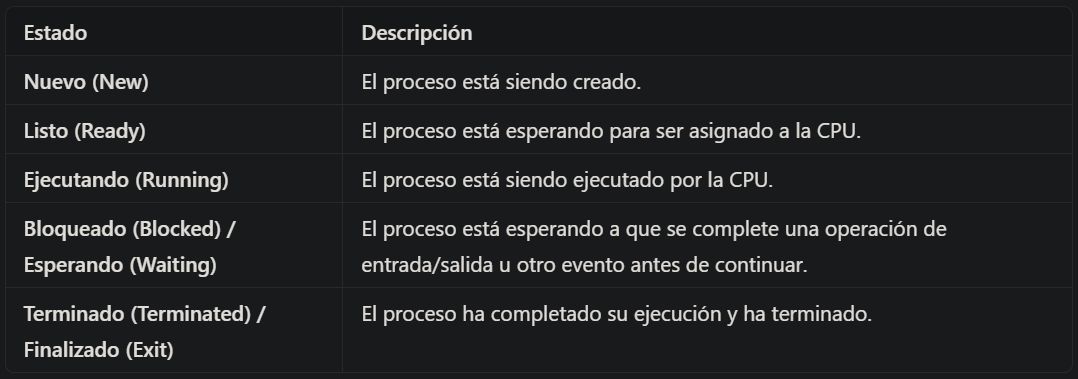
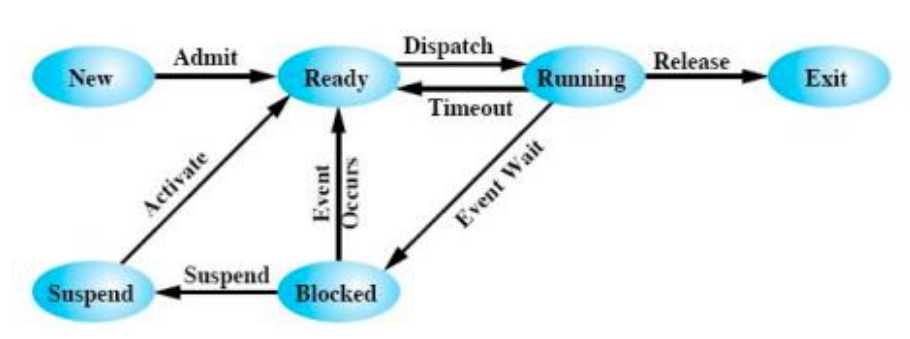
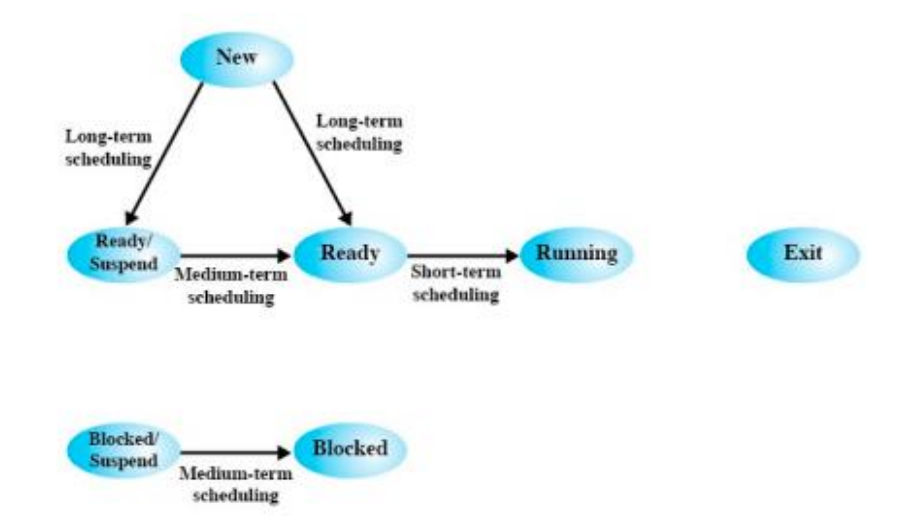
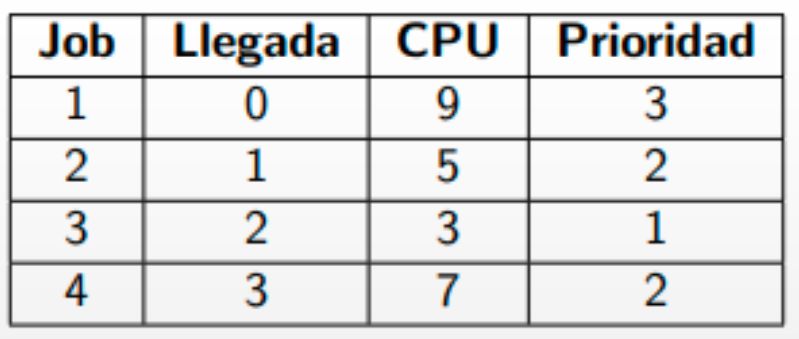
**Comunicación padre-hijo:** En la mayoría de los casos, las pipes se utilizan en escenarios en los que un proceso **padre** y su **proceso hijo** se comunican entre sí. El proceso padre puede enviar información al hijo a través del pipe, o viceversa.

* + La información mínima que el sistema operativo debe tener sobre un proceso es el PID, el PPID, la Ubicación del mismo en memoria, los Recursos Asignados (espacio de memoria, descriptores de archivos, etc.), el Accounting, la Entrada y Salida, el Contexto de Ejecución (valores de los registros de la CPU), y la Planificación del mismo (prioridad, estado, tiempo consumido, etc.). Toda esta información se almacena en una estructura llamada PCB (Process Control Block).

Características de la PCB:

* Existe una por proceso.
* Es lo primero que se crea cuando un proceso es creado y también es lo último que se borra cuando el proceso termina.
* Tiene referencias a memoria.
* Se puede pensar como un gran registro que almacena toda esta información.
  + Un **proceso CPU bound** es aquel cuyo rendimiento está principalmente limitado por la velocidad de la **unidad central de procesamiento (CPU)**. Estos procesos pasan la mayor parte de su tiempo realizando cálculos intensivos o ejecutando operaciones que requieren una gran cantidad de procesamiento, como algoritmos complejos, simulaciones, compresión de datos, etc. Estos procesos son llamados "limitados por la CPU" porque el factor que más impacta su rendimiento es la velocidad de la CPU, no la velocidad del acceso a memoria o de las operaciones de entrada/salida (I/O).

Un **proceso I/O bound** es aquel cuyo rendimiento está principalmente limitado por la velocidad de las operaciones de **entrada/salida (I/O)**, como leer o escribir datos en un disco, esperar por la entrada del usuario, acceder a la red, o interactuar con otros dispositivos periféricos (por ejemplo, impresoras, discos duros, etc.). Estos procesos pasan mucho tiempo esperando a que las operaciones de I/O se completen, y no requieren mucha CPU en comparación con los procesos **CPU Bound**.

* + 
  + 
  + 
* 3)
  + 

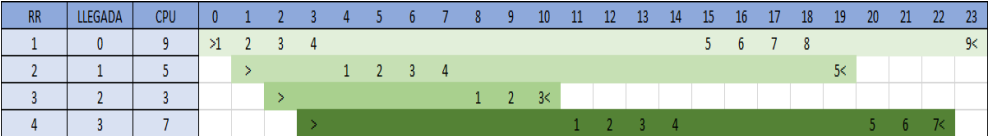
**FCFS:**

* Cuando hay que elegir un proceso para ejecutar, se elige al más viejo.
* No favorece a ningún tipo de procesos.
* Los CPU Bound terminan al comenzar su primera ráfaga mientras que los I/O Bound no.
* Es no apropiativo.
* Orden de ejecución: Job 1, Job 2, Job 3, Job 4.

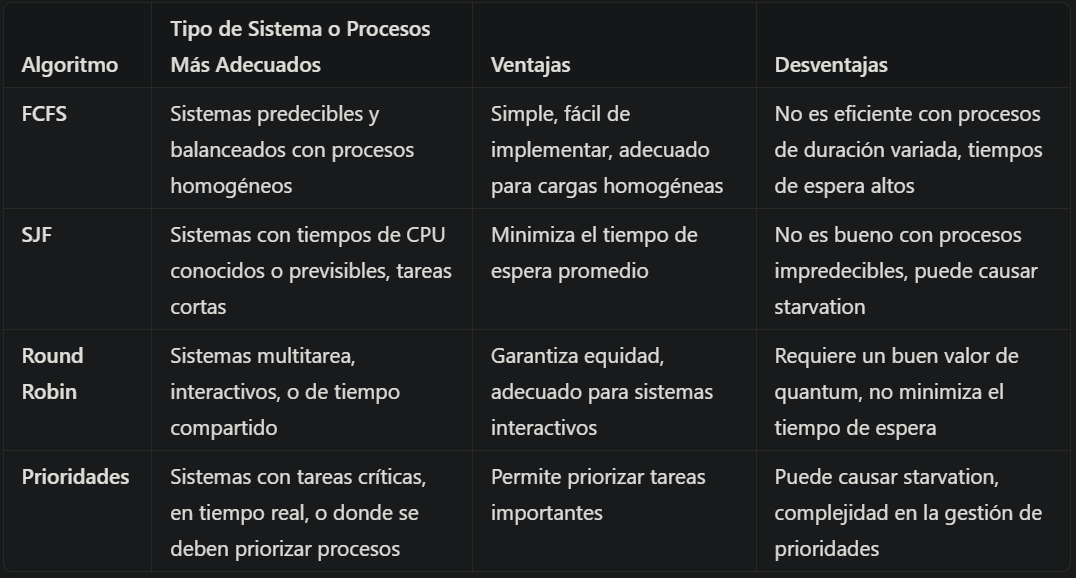
**SJF:**

* Selecciona el proceso con la ráfaga más corta.
* Es no apropiativo.
* Los procesos cortos y los I/o Bound se ven beneficiados.
* Los procesos largos pueden sufrir starvation (inanición) ya que pueden pasar mucho tiempo sin ser seleccionados.
* Orden de ejecución: Job 1, Job 3, Job 2, Job 4.

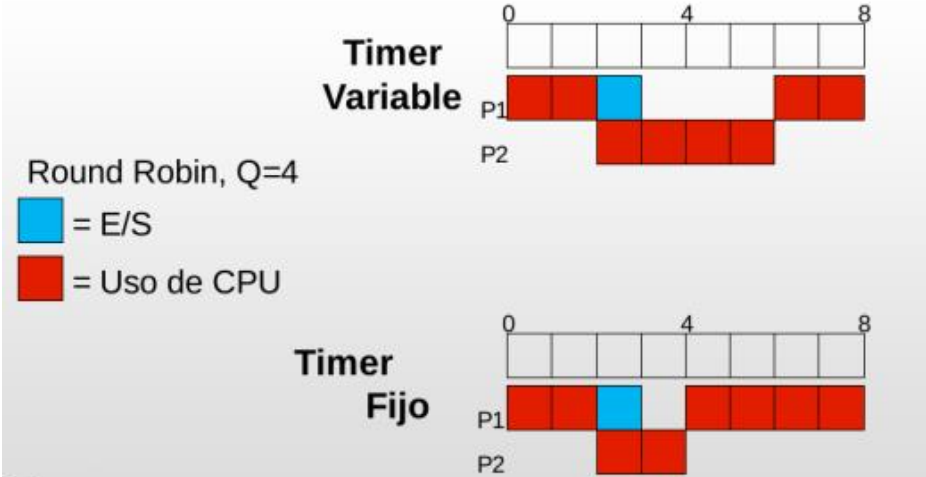
**Round Robin:**

* Política basada en el uso de un reloj (Quantum).
* Cuando un proceso es expulsado de la CPU es colocado al final de la cola de listos y se selecciona otro generando un FIFO circular.
* Existe un “contador” que indica las unidades de CPU en las que el proceso se ejecutó.
* Cuando este llega a 0 el proceso es expulsado.
* El “contador” puede ser Global o Local (PCB).
* Se puede tener 2 variantes con respecto al valor inicial del “contador” cuando un proceso es asignado a la CPU: Timer Variable o Timer Fijo.
* Orden de ejecución con un Quantum = 4 y Timer Variable:
* 

**Prioridades:**

* Cada proceso tiene un valor que representa su prioridad, mientras menor sea su valor, mayor prioridad.
* Se selecciona al proceso con mayor prioridad de la cola de listos. Puede ser apropiativo o no.
* Existe una cola de listos para cada nivel de prioridad.
* Los procesos de baja prioridad pueden sufrir starvation (inanición).
* Orden de ejecución No Apropiativo: Job 1, Job 3, Job 2, Job 4.
  + Únicamente el Round Robin necesita la definición del tamaño del Quantum.
  + Algoritmos:
* FIFO/FCFS: No favorece a ningún tipo de procesos.
* SJF: Favorece a los I/O Bound y a los procesos cortos.
* Round Robin: Favorece más a los I/O Bound.
* Prioridades: Favorece a los de mayor prioridad.
  + 
* 4)
  + **Timer Fijo:**
* **Definición**: En la variante de **timer fijo**, el **quantum** de tiempo asignado a cada proceso es constante. Es decir, todos los procesos reciben la misma cantidad de tiempo para ejecutarse.
* **Cómo funciona**: Cada vez que un proceso es asignado a la CPU, se le da exactamente el mismo tiempo (el quantum) para ejecutarse. Si el proceso no termina dentro de ese tiempo, se le suspende y se le coloca al final de la cola de procesos listos. Si el proceso termina antes de que se acabe el quantum, el siguiente proceso se ejecuta de inmediato.

**Timer Variable:**

* **Definición**: En la variante de **timer variable**, el **quantum** no es constante. En lugar de usar un valor fijo para todos los procesos, el tiempo de quantum puede variar.
* **Cómo funciona**: En algunos sistemas con **timer variable**, si un proceso no llega a usar todo su Quantum, el sobrante lo usará el siguiente proceso a ejecutarse y cuando el Quantum llegue a cero, se reasignará su valor.
  + 
  + Para el Timer Variable, cada proceso debería almacenar la información de su Quantum en su PCB, mientras que con Timer Fijo, la información se debe almacenar en una estructura o espacio que sea accesible directamente por el sistema operativo.
* 5)
  + Resuelto en los ejercicios .xlsx
* 6)
  + Resuelto en los ejercicios .xlsx
  + El algoritmo con mejores promedios fue el SJF, mientras que el algoritmo con peor promedio fue el Round Robin de Timer Variable con Quantum 1.
  + Respecto al valor del Quantum podemos decir que mientras más chico sea, más cambios de contexto habrá, haciendo que los promedios se disparen, y cuanto más grande sea, se comporta de manera similar al FCFS disminuyendo los promedios.
  + Utilizaría un Quantum alto para manejar lotes de procesos CPU Bound ya que estos en lo general ocupan por un largo tiempo los recursos de la CPU haciendo que seguramente se amolden bien al valor del Quantum, disminuyendo las tasas de tiempo de retorno de todos estos procesos, factor que toma mucha importancia cuando hablamos de procesos CPU Bound. La desventaja de esto sería que si en el lote de procesos nos encontramos procesos cortos o por ejemplo procesos I/O Bound que por lo general no hacen mucho consumo de la CPU, con un Quantum alto podríamos generar que las tasas de tiempo de espera se disparen, haciendo que los procesos 24 cortos tengan que esperar muchos ciclos de CPU para poder ejecutarse.
* 7)
  + Resuelto en los ejercicios .xlsx
  + Genera un buen promedio de tiempo de espera, por lo tanto, favorece ventajosamente a procesos I/O Bound ya que estos dependen del tiempo de espera, también los procesos cortos se verían beneficiados.
* 8)
  + Resuelto en los ejercicios .xlsx
  + Las ventajas que les puedo dar a estos algoritmos es que para tareas críticas del sistema operativo que necesiten ser ejecutadas inmediatamente, van a ser muy buenos, además de que nos brindan una mayor adaptabilidad en nuestro sistema si decidimos llevar una organización por prioridades. Los casos donde los utilizaría son justamente los mencionados en las ventajas.
* 9)
  + La **inanición** (o **starvation** en inglés) es una situación en la que un proceso dentro de un sistema operativo no puede acceder a los recursos que necesita para continuar su ejecución, quedando bloqueado o en espera indefinida. Esto sucede cuando otros procesos continuamente acaparan esos recursos o tienen prioridad sobre el proceso afectado, lo que provoca que el proceso nunca avance o se complete.
  + De los algoritmos vistos los que pueden generar inanición serían el SJF, el STRF y el de Prioridades. Los 2 primeros generándola para procesos largos y el tercero en procesos de baja prioridad.
  + Existen varias técnicas para evitar la Inanición:
* **Envejecimiento (Aging):** Consiste en aumentar la prioridad de un proceso a medida que pasa el tiempo sin ser seleccionado para su ejecución.
* **Establecimiento de límites de espera:** Consiste en establecer límites de tiempo máximo para la espera de un proceso en la cola de listos, haciendo que si llega a este límite, pase a ejecutarse.
* **Prioridades dinámicas:** Podemos tener sistemas que ajusten las prioridades de los procesos dinámicamente según el comportamiento y las necesidades del proceso.
* 10)
  + Resuelto en los ejercicios .xlsx
* 11)
  + Round Robin:
* El algoritmo Round Robin para procesos que son CPU Bound podría presentar como desventajas que los tiempos de espera se vuelven bastante grandes, que se genera cierto desperdicio de tiempo de CPU si utilizamos Timer Variable y un proceso termina antes de terminar su Quantum, y también podemos incluir como desventaja que al usar este algoritmo en procesos CPU Bound se puede producir un Overhead de context switch que reduzca la eficiencia del sistema. En cuanto a los procesos I/O Bound, este algoritmo es más adecuado para estos tipos de procesos pero presenta ciertas desventajas a su vez como pueden ser problemas de inanición, ineficiencia en la utilización de la CPU y retrasos en la finalización de estos procesos ya que pueden ser interrumpidos por el context switch.
  + SRTF:
* Este algoritmo beneficia a los procesos I/O Bound así que no le veo muchas desventajas pero para los procesos CPU Bound creo que este algoritmo podría generar inanición de procesos con larga duración y también podría generar una sobrecarga de cambios de contexto si la mayoría de procesos que están en la cola de listos tienen un tiempo de ejecución similar.
* 12)
  + Resuelto en los ejercicios .xlsx
* 13)
  + Sí, esto podría ocurrir si un proceso termina su ejecución en una ráfaga de CPU antes de que se termine su Quantum.
* 14)

| Nro de Ráfaga (n) | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Valor Estimado (Sn) | 10 | 6 | 5 | 5.3 | 5 | 6.6 | 7.6 |
| Duración de la Ráfaga (Tn) | 6 | 4 | 6 | 4 | 13 | 13 | 13 |

* + Para valores de delta que tienden a 1, lo que ocurre es que el primer término de la fórmula toma más valor, mientras que si el delta tiende a 0, toma más valor el segundo término de la fórmula.

| Nro de Ráfaga (n) | α | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Valor Estimado (Sn) | 0,2 | 10 | 9,2 | 8,16 | 7,72 | 6,97 | 8,17 | 9,13 |
| 0,5 | 10 | 8 | 6 | 6 | 5 | 9 | 11 |
| 0,8 | 10 | 6,8 | 4,56 | 5,71 | 4,34 | 11,26 | 12,65 |
| Duración de la Ráfaga (Tn) | ~ | 6 | 4 | 6 | 4 | 13 | 13 | 13 |

* + La que más se asemeja es la del delta = 0.8.
* 15)
  + Para procesos Interactivos usaría Round Robin y para procesos Batch usaría FCFS.
* 16)
  + Yo utilizaría un algoritmo por Prioridades con Aging.
* 17)
  + Resuelto en los ejercicios .xlsx
* 18)
  + Manejaría 3 colas cada una con un nivel de prioridad, los procesos más largos tendrían menor prioridad y los más cortos tendrían mayor prioridad, cada cola individualmente la manejaría usando un RR con un Quantum chico que favorece a los procesos de menor duración y para solucionar el problema de inanición de los procesos más largos, usaría un algoritmo por prioridades con aging para que los procesos vayan escalando en prioridad a medida que los procesos más chicos van terminando su ejecución.
* 19)
  + Yo creo que las actividades que tendrán más prioridad son las que requieran una interacción con el usuario o las que conllevan que se use poco la CPU ya que si el scheduler selecciona estas actividades podría minimizar el tiempo de espera para esos procesos y maximizar el poco uso que estos procesos puedan requerir de CPU, dejándole paso a procesos CPU Bound de los cuáles el usuario promedio no se entera de su existencia.
  + Los procesos que se verían favorecidos serían los procesos I/O Bound ya que con estos el usuario interactúa más.
  + Yo creo que el favorecer o no lo determina el hecho de cómo se configure el Round Robin, es decir, tener en cuenta el tamaño del Quantum ya que si es muy pequeño puede generar un overhead de context switch, mientras que si es muy grande puede disminuir la capacidad de respuesta del sistema. Como también lo determina el número de niveles en las colas multinivel y la configuración de los 34 niveles de prioridad. Pero en general, si se configura bien, el usar Round Robin puede proporcionar equidad, prevención de inanición, respuestas rápidas del sistema y una adaptabilidad a diversas cargas de trabajo.
* 20)
  + Cortos acotados por CPU se ven beneficiados. Cortos acotados por E/S se ven beneficiados. Largos acotados por CPU se ve penalizados siempre que haya procesos más cortos antes que estos. Largos acotados por E/S se ven penalizados por lo mismo que los anteriores más el tiempo de espera para usar la E/S.
  + Tomándolo como mientras menor sea el tiempo transcurrido desde la última operación de entrada y salida mayor va a ser la prioridad, creo que los únicos procesos que se beneficiarían serían los que usan la E/S ya que serían los únicos capaces de escalar en prioridad.
* 21)
  + Esto ocurre porque va a llegar un punto donde el Quantum sea tan grande que todos los procesos terminen o salgan de la CPU sin ser expulsados por el algoritmo, eliminando toda la rotación de procesos que plantea el Round Robin convirtiendo la cola en una circular del estilo FIFO.
* 22)
  + Con los Homogéneos o los Multiprocesador fuertemente acoplados.
  + Esto significa que todos los procesadores o núcleos de la CPU tienen la misma prioridad y pueden ejecutar cualquier tarea. No existe una distinción jerárquica entre los procesadores en términos de capacidad de procesamiento o acceso a recursos del sistema.
  + Esto significa que establece una jerarquía entre los procesadores para coordinar y distribuir las tareas, el Maestro tiene el control central y toma decisiones críticas, mientras que los esclavos ejecutan las tareas asignadas.
* 23)
  + “Si se asume que la arquitectura del multiprocesador es uniforme, en el sentido de que ningún procesador tiene una ventaja física particular con respecto al acceso a memoria principal o a dispositivos de E/S, entonces el enfoque más simple de la planificación consiste en tratar cada proceso como un recurso colectivo y asignar procesos a procesadores por demanda. Surge la cuestión de si la asignación debería ser estática o dinámica. Si un proceso se vincula permanentemente a un procesador desde su activación hasta que concluye, entonces se mantiene una cola a corto plazo dedicada por cada procesador. Una ventaja de esta estrategia es que puede haber menos sobrecarga en la función de planificación, dado que la asignación a un procesador se realiza una vez y para siempre. Asimismo, el uso de procesadores dedicados permite una estrategia conocida como planificación de grupo o pandilla, como se verá más adelante. Una desventaja de la asignación estática es que un procesador puede estar ocioso, con su cola vacía, mientras otro procesador tiene trabajo acumulado. Para evitar esta situación, puede utilizarse una cola común. Todos los procesos van a una cola global y son planificados sobre cualquier procesador disponible. Así, a lo largo de la vida de un proceso, puede ser ejecutado en diferentes procesadores en diferentes momentos.”

Sistemas Operativos 5ta edición, William Stallings.

* 24)
  + **Huella de un proceso en un procesador:**

Es el estado que un proceso va dejando en la caché de un procesador, dejando información como la cantidad de memoria RAM que ocupa, la cantidad de CPU que consume, los archivos y dispositivos que utiliza, entre otras cosas.

* + **Afinidad con un Procesador:**

Es la preferencia de un proceso para ejecutarse en un procesador específico.

* + El por qué se combina con el concepto de huella ya que si un proceso tiene mayor afinidad con un procesador, cada vez que se ejecute irá dejando su huella allí por lo tanto es más probable que el procesador tenga almacenada información de ese proceso en su caché, reduciendo el tiempo necesario para la ejecución del proceso en cuestión.
  + Sí, en ambos sistemas operativos es posible que el usuario pueda cambiar la afinidad de un proceso.
  + Load Balancing:

Es una técnica utilizada para distribuir de manera equitativa la carga de trabajo entre múltiples recursos. Tiene como objetivo principal mejorar la eficiencia, disponibilidad y rendimiento del sistema al evitar la sobrecarga de un recurso particular y aprovechar al máximo los recursos disponibles. Es fundamental en entornos donde la disponibilidad y el rendimiento son críticos.

* + Los dos conceptos son estrategias importantes en la administración de procesos y recursos en un sistema. La afinidad se ve más centrada en la asignación específica de procesos a recursos, mientras que el balanceo de carga busca la distribución equitativa de la carga entre los recursos. Ambos son cruciales para el rendimiento y eficiencia de un sistema, pero pueden entrar en conflicto si no se gestionan adecuadamente, por lo tanto es necesario buscar un equilibrio entre estos conceptos para optimizar el funcionamiento del sistema.
* 25)
  + Resuelto en los ejercicios .xlsx
  + La alternativa planteada en b)