**1)**

**(a) Programa y Proceso**

* **Programa**: Conjunto de instrucciones almacenadas que pueden ser ejecutadas por un sistema.
* **Proceso**: Ejecución activa de un programa, incluyendo su estado y recursos asignados.

**(b) Tiempo de Retorno (TR) y Tiempo de Espera (TE) para un Job**

* **Tiempo de Retorno (TR)**: Tiempo total desde la llegada del job al sistema hasta su finalización. TR=TiempodeFinalizacioˊn−TiempodeLlegadaTR = Tiempo de Finalización - Tiempo de LlegadaTR=TiempodeFinalizacioˊn−TiempodeLlegada
* **Tiempo de Espera (TE)**: Tiempo que un job pasa en las colas del sistema sin ser ejecutado. TE=TR−TiempodeEjecucioˊnTE = TR - Tiempo de EjecuciónTE=TR−TiempodeEjecucioˊn

**(c) Tiempo Promedio de Retorno (TPR) y Tiempo Promedio de Espera (TPE) para un lote de JOBS**

* **TPR**: Promedio de los tiempos de retorno para todos los jobs en un lote. TPR=∑TRNTPR = \frac{\sum TR}{N}TPR=N∑TR​
* **TPE**: Promedio de los tiempos de espera para todos los jobs en un lote. TPE=∑TENTPE = \frac{\sum TE}{N}TPE=N∑TE​

**(d) ¿Qué es el Quantum?**

* Es el tiempo máximo asignado a un proceso en el algoritmo de *Round Robin*. Cuando se agota, el proceso se interrumpe y se pasa al siguiente.

**(e) Algoritmo Apropiativo o No Apropiativo**

* **Apropiativo (Preemptive)**: Permite interrumpir un proceso en ejecución para dar paso a otro de mayor prioridad.
* **No Apropiativo (Non-Preemptive)**: Un proceso en ejecución no puede ser interrumpido hasta que termine o libere la CPU.

**(f) Tareas de los Schedulers**

1. **Short Term Scheduler**: Selecciona cuál proceso se ejecutará en la CPU en un momento dado.
2. **Long Term Scheduler**: Decide qué procesos ingresan al sistema para ser ejecutados, controlando la carga.
3. **Medium Term Scheduler**: Suspende o reanuda procesos para optimizar el uso de recursos.

**(g) Tareas del Dispatcher**

* Transfiere el control de la CPU al proceso seleccionado por el *Short Term Scheduler*, realizando:
  + Cambio de contexto.
  + Paso a modo usuario.
  + Salto a la dirección de inicio del programa.

**2)**

### (a) Comandos y su utilidad

#### i. top

* Muestra en tiempo real los procesos activos en el sistema y su consumo de recursos como CPU, memoria y tiempo de ejecución.

#### ii. htop

* Herramienta interactiva similar a top, pero con interfaz más amigable. Permite gestionar procesos directamente con el teclado o ratón.

#### iii. ps

* Muestra información sobre los procesos en ejecución. Puede listar procesos específicos o todos los procesos en el sistema.

#### iv. pstree

* Muestra los procesos en forma de árbol, indicando las relaciones jerárquicas entre ellos (procesos padre e hijo).

#### v. kill

* Envía una señal a un proceso para controlarlo. Comúnmente se usa para finalizar procesos (señal SIGKILL).

#### vi. pgrep

* Busca procesos por nombre o atributos específicos y devuelve sus IDs de proceso (PIDs).

#### vii. killall

* Finaliza todos los procesos que coinciden con un nombre dado.

#### viii. renice

* Cambia la prioridad de ejecución de un proceso en ejecución para aumentar o disminuir el uso de CPU.

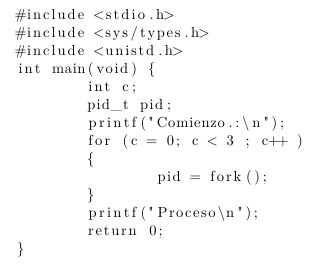
#### ix. xkill

* Herramienta gráfica que permite cerrar ventanas de aplicaciones haciendo clic en ellas. Debe estar instalada previamente.

#### x. atop

* Monitoriza procesos en tiempo real y registra el uso de recursos del sistema como CPU, memoria, disco y red, con un enfoque en análisis posterior.

**B)**



#### Código explicado:

1. **Bibliotecas**: Se incluyen las necesarias para manejo de procesos (unistd.h) y operaciones estándar (stdio.h).
2. **Variables**:
   * c: Contador del bucle.
   * pid: Almacena el ID del proceso devuelto por fork().
3. **Función principal**:
   * Imprime "Comienzo".
   * Un bucle for ejecuta fork() tres veces. Cada llamada a fork() crea un proceso hijo.
   * Después del bucle, cada proceso (padres e hijos) imprime "Proceso".

#### ¿Qué hace fork()?

* Crea un nuevo proceso hijo que es una copia del proceso padre.
* Devuelve:
  + 0 en el proceso hijo.
  + ID del hijo en el proceso padre.

#### Cálculo del número de procesos creados:

Cada llamada a fork() duplica el número de procesos en ejecución. Con 3 iteraciones del bucle, el número de procesos es: Nuˊmero de procesos=2n\text{Número de procesos} = 2^nNuˊmero de procesos=2n Donde nnn es el número de iteraciones. Para n=3n = 3n=3, hay: 23=82^3 = 823=8

#### i. ¿Cuántas líneas con "Proceso" aparecen?

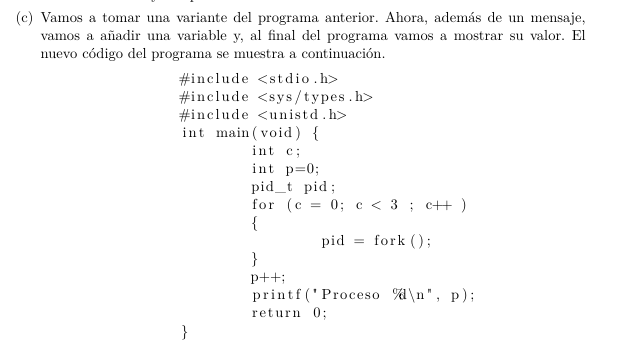
* Cada uno de los 8 procesos imprime "Proceso".
* Aparecen **8 líneas con "Proceso"**.

#### ii. ¿El número de líneas es igual al número de procesos que han estado en ejecución?

* Sí, cada proceso en ejecución imprime una línea con "Proceso".

#### Ejecución con diferentes valores de n:

* Si cambias el valor del bucle for a n=4n = 4n=4: 24=162^4 = 1624=16 Se crearán 16 procesos y aparecerán 16 líneas.
* Para n=5n = 5n=5: 25=322^5 = 3225=32 Se crearán 32 procesos y aparecerán 32 líneas.



### Análisis del programa modificado

#### Código explicado:

1. **Variables**:
   * c: Contador del bucle.
   * p: Inicializado en 0 y luego incrementado.
   * pid: Almacena el valor de retorno de fork().
2. **Bucle**:
   * Ejecuta fork() tres veces, creando procesos hijos.
   * Cada proceso hijo es una copia del padre, con su propio espacio de memoria (variables independientes).

#### Incremento de p:

* La variable p se incrementa una vez después de las llamadas a fork(). Como cada proceso tiene su copia de p, los incrementos no afectan a los otros procesos.

#### i. ¿Qué valores se muestran por consola?

* Cada proceso imprime el valor de p después de incrementarlo.
* Inicialmente, cada proceso comienza con p = 0 y luego lo incrementa a p = 1.
* Así, cada proceso imprimirá: Proceso 1\text{Proceso 1}Proceso 1

#### ii. ¿Todas las líneas tendrán el mismo valor o algunas tendrán valores distintos?

* Todas las líneas tendrán el mismo valor (p = 1), porque cada proceso independiente comienza con p = 0 y lo incrementa localmente.

#### iii. ¿Cuál es el valor (o valores) que aparece?

* El valor que aparece en cada línea es **1**.

#### Observaciones al ejecutar:

* **Incremento fuera del bucle**: Cada proceso imprimirá **1**.
* **Incremento dentro del bucle**: Los valores impresos serán diferentes (por ejemplo, 1, 2, 3), dependiendo del número de veces que cada proceso haya ejecutado p++. Esto refleja cómo cada proceso gestiona su copia independiente de p.

### (d) Comunicación entre procesos: ****Pipes****

#### i. Comunicación entre procesos a través de pipes

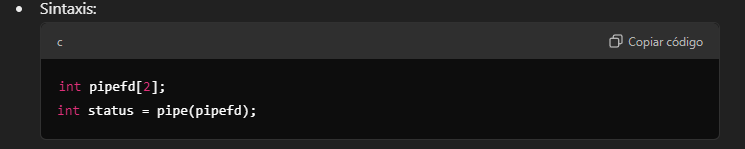
* Un **pipe** es un mecanismo de comunicación entre procesos (Inter-Process Communication o IPC).
* Permite que un proceso escriba datos en un extremo del pipe, mientras que otro proceso los lee desde el otro extremo.
* Los pipes son unidireccionales: la comunicación es de un único sentido (de escritura a lectura).

**Ejemplo de uso típico**:

* Un proceso padre crea un pipe antes de realizar un fork().
* Tanto el proceso padre como el hijo pueden usar los extremos del pipe para intercambiar datos.

#### ii. ¿Cómo se crea un pipe en C?

* La función pipe() se usa para crear un pipe.



* + pipefd[0]: Descriptor para el extremo de lectura del pipe.
  + pipefd[1]: Descriptor para el extremo de escritura del pipe.

#### iii. Parámetro necesario para la creación de un pipe

* **Parámetro**: Un arreglo de dos enteros (int pipefd[2]).
* **Uso**:
  + **pipefd[0]**: Utilizado para leer datos del pipe.
  + **pipefd[1]**: Utilizado para escribir datos en el pipe.

**Funcionamiento**:

1. Los datos escritos en pipefd[1] por un proceso están disponibles para ser leídos desde pipefd[0] por otro proceso.
2. Si un extremo del pipe no es necesario en un proceso (por ejemplo, un proceso que solo escribe), se cierra con close().

### (d) Comunicación entre procesos: ****Pipes****

#### i. Comunicación entre procesos a través de pipes

* Un **pipe** es un mecanismo de comunicación entre procesos (Inter-Process Communication o IPC).
* Permite que un proceso escriba datos en un extremo del pipe, mientras que otro proceso los lee desde el otro extremo.
* Los pipes son unidireccionales: la comunicación es de un único sentido (de escritura a lectura).

**Ejemplo de uso típico**:

* Un proceso padre crea un pipe antes de realizar un fork().
* Tanto el proceso padre como el hijo pueden usar los extremos del pipe para intercambiar datos.

#### ii. ¿Cómo se crea un pipe en C?

* La función pipe() se usa para crear un pipe.
* Sintaxis:

c

Copiar código

int pipefd[2];

int status = pipe(pipefd);

* + pipefd[0]: Descriptor para el extremo de lectura del pipe.
  + pipefd[1]: Descriptor para el extremo de escritura del pipe.

#### iii. Parámetro necesario para la creación de un pipe

* **Parámetro**: Un arreglo de dos enteros (int pipefd[2]).
* **Uso**:
  + **pipefd[0]**: Utilizado para leer datos del pipe.
  + **pipefd[1]**: Utilizado para escribir datos en el pipe.

**Funcionamiento**:

1. Los datos escritos en pipefd[1] por un proceso están disponibles para ser leídos desde pipefd[0] por otro proceso.
2. Si un extremo del pipe no es necesario en un proceso (por ejemplo, un proceso que solo escribe), se cierra con close().

### Ejemplo básico en C:



#### Explicación del ejemplo:

1. **Creación del pipe**:
   * pipe(pipefd) crea un pipe con descriptores pipefd[0] (lectura) y pipefd[1] (escritura).
2. **División de procesos**:
   * Proceso padre escribe en pipefd[1].
   * Proceso hijo lee desde pipefd[0].
3. **Cierre de extremos no usados**:
   * Cada proceso cierra el extremo del pipe que no necesita para evitar bloqueos.

### (iv) ¿Qué tipo de comunicación es posible con pipes?

Los pipes permiten una **comunicación unidireccional** entre procesos relacionados, generalmente entre un **proceso padre** y sus **procesos hijos**. Los datos fluyen en una sola dirección: de un extremo de escritura a un extremo de lectura.

#### Tipos de comunicación con pipes:

1. **Unidireccional**: Solo se puede escribir en un extremo y leer en el otro.
2. **Relación padre-hijo**: Los pipes básicos requieren que los procesos compartan un ancestro común.

Para lograr **bidireccionalidad**, se deben usar **dos pipes** o utilizar alternativas como sockets o named pipes (FIFOs).

**(e) Información mínima que el SO debe tener sobre un proceso**

El sistema operativo almacena información sobre cada proceso en una estructura llamada **PCB** (*Process Control Block*). Los datos mínimos incluyen:

1. **Identificación del proceso**:
   * PID (Process ID).
   * Identificadores del proceso padre e hijos.
2. **Estado del proceso**:
   * Listo, en ejecución, bloqueado, etc.
3. **Contexto de CPU**:
   * Registros, contador de programa, pila.
4. **Memoria**:
   * Segmentos de código, datos, pila y heap.
   * Tabla de páginas o segmentos asignados.
5. **Información de I/O**:
   * Archivos abiertos.
   * Dispositivos asociados.
6. **Prioridad y planificación**:
   * Prioridad del proceso.
   * Información de tiempos (CPU y E/S).

El PCB es almacenado en una estructura propia del kernel y se accede mediante colas de planificación o listas de estado.

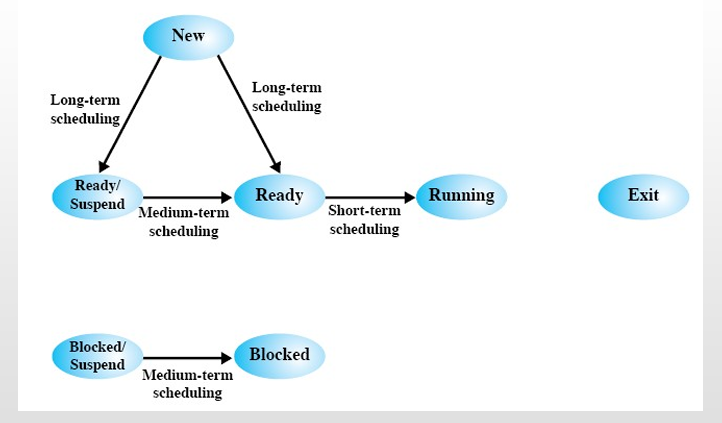
**(f) ¿Qué significa que un proceso sea *CPU Bound* y *I/O Bound*?**

1. **CPU Bound**:
   * La mayor parte del tiempo, el proceso utiliza la CPU para realizar cálculos intensivos.
   * Ejemplo: Simulaciones científicas, cifrado.
2. **I/O Bound**:
   * El proceso pasa más tiempo esperando operaciones de entrada/salida (I/O) que ejecutando cálculos en la CPU.
   * Ejemplo: Procesamiento de archivos, servidores de bases de datos.

**(g) Estados posibles de un proceso**

1. **Nuevo (New)**: El proceso está siendo creado.
2. **Listo (Ready)**: Está preparado para ejecutarse, esperando asignación de CPU.
3. **Ejecutando (Running)**: Está siendo ejecutado por la CPU.
4. **Bloqueado (Blocked/Waiting)**: Espera un evento externo, como una operación de E/S.
5. **Terminado (Terminated)**: Ha finalizado su ejecución.

**(h) Transiciones entre estados**

****

#### Explicación de las transiciones:

1. **New → Ready**: Proceso creado y cargado en memoria, esperando CPU.
2. **Ready → Running**: Scheduler asigna CPU al proceso.
3. **Running → Ready**: Proceso es despojado de la CPU (time slicing o preempción).
4. **Running → Blocked**: Proceso espera por un evento externo (E/S).
5. **Blocked → Ready**: Evento esperado ocurre, proceso puede continuar.
6. **Running → Terminated**: Proceso finaliza su ejecución.

### (iv) ¿Qué tipo de comunicación es posible con pipes?

Los pipes permiten una **comunicación unidireccional** entre procesos relacionados, generalmente entre un **proceso padre** y sus **procesos hijos**. Los datos fluyen en una sola dirección: de un extremo de escritura a un extremo de lectura.

#### Tipos de comunicación con pipes:

1. **Unidireccional**: Solo se puede escribir en un extremo y leer en el otro.
2. **Relación padre-hijo**: Los pipes básicos requieren que los procesos compartan un ancestro común.

Para lograr **bidireccionalidad**, se deben usar **dos pipes** o utilizar alternativas como sockets o named pipes (FIFOs).

### (e) Información mínima que el SO debe tener sobre un proceso

El sistema operativo almacena información sobre cada proceso en una estructura llamada **PCB** (Process Control Block). Los datos mínimos incluyen:

1. **Identificación del proceso**:
   * PID (Process ID).
   * Identificadores del proceso padre e hijos.
2. **Estado del proceso**:
   * Listo, en ejecución, bloqueado, etc.
3. **Contexto de CPU**:
   * Registros, contador de programa, pila.
4. **Memoria**:
   * Segmentos de código, datos, pila y heap.
   * Tabla de páginas o segmentos asignados.
5. **Información de I/O**:
   * Archivos abiertos.
   * Dispositivos asociados.
6. **Prioridad y planificación**:
   * Prioridad del proceso.
   * Información de tiempos (CPU y E/S).

El PCB es almacenado en una estructura propia del kernel y se accede mediante colas de planificación o listas de estado.

### (f) ¿Qué significa que un proceso sea CPU Bound y I/O Bound?

1. **CPU Bound**:
   * La mayor parte del tiempo, el proceso utiliza la CPU para realizar cálculos intensivos.
   * Ejemplo: Simulaciones científicas, cifrado.
2. **I/O Bound**:
   * El proceso pasa más tiempo esperando operaciones de entrada/salida (I/O) que ejecutando cálculos en la CPU.
   * Ejemplo: Procesamiento de archivos, servidores de bases de datos.

### (g) Estados posibles de un proceso

1. **Nuevo (New)**: El proceso está siendo creado.
2. **Listo (Ready)**: Está preparado para ejecutarse, esperando asignación de CPU.
3. **Ejecutando (Running)**: Está siendo ejecutado por la CPU.
4. **Bloqueado (Blocked/Waiting)**: Espera un evento externo, como una operación de E/S.
5. **Terminado (Terminated)**: Ha finalizado su ejecución.

### (h) Transiciones entre estados

#### Diagrama:

plaintext

Copiar código

New

↓

Ready ←→ Running → Terminated

↑ ↓

Blocked ←───

#### Explicación de las transiciones:

1. **New → Ready**: Proceso creado y cargado en memoria, esperando CPU.
2. **Ready → Running**: Scheduler asigna CPU al proceso.
3. **Running → Ready**: Proceso es despojado de la CPU (time slicing o preempción).
4. **Running → Blocked**: Proceso espera por un evento externo (E/S).
5. **Blocked → Ready**: Evento esperado ocurre, proceso puede continuar.
6. **Running → Terminated**: Proceso finaliza su ejecución.

### (i) ¿Qué scheduler realiza las transiciones?

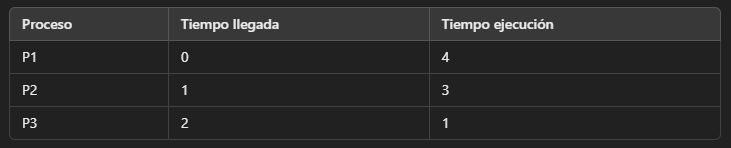
1. **Long-Term Scheduler**:
   * Gestiona la transición de **New → Ready** al decidir qué procesos cargar en memoria.
2. **Short-Term Scheduler**:
   * Maneja las transiciones entre:
     + **Ready → Running**.
     + **Running → Ready**.
     + **Running → Terminated**.
3. **Medium-Term Scheduler** (si existe):
   * Gestiona **Suspensión** para procesos bloqueados o con baja prioridad, retirándolos de memoria y devolviéndolos al estado **Ready** cuando sea necesario.

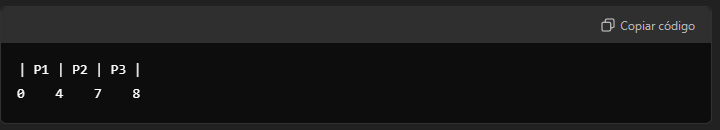
### (a) Funcionamiento mediante ejemplo

#### 1. ****FCFS (First Come First Served)****:

* Los procesos se atienden en el orden de llegada.

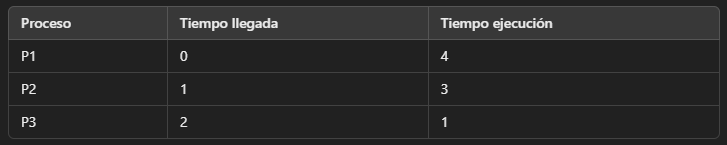
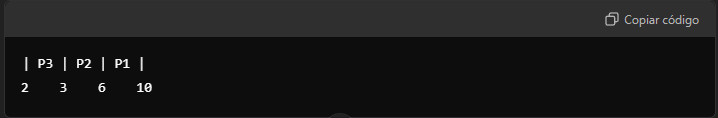
**Ejemplo**:  
Procesos con tiempos de llegada y ejecución:

****

**Orden de ejecución**: P1 → P2 → P3  
**Gantt**: 

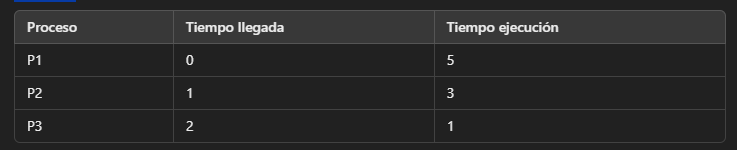
#### 2. ****SJF (Shortest Job First)****:

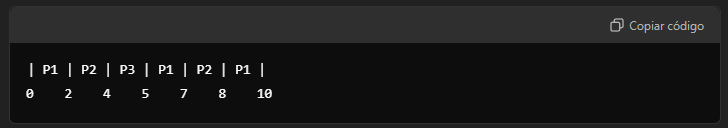
* El proceso con menor tiempo de ejecución se ejecuta primero.

**Ejemplo**: **Orden de ejecución**: P3 → P2 → P1  
**Gantt**: 

#### 3. ****Round Robin (RR)****:

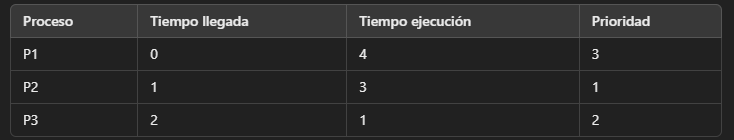
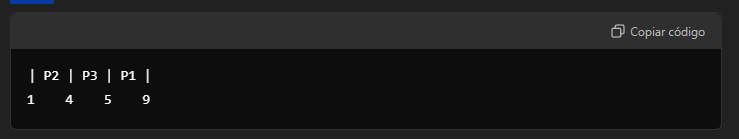
* Cada proceso obtiene un intervalo fijo de tiempo (quantum) para ejecutarse. Si no termina, vuelve a la cola.

**Ejemplo**: 

**Quantum = 2**  
**Orden de ejecución**: P1 → P2 → P3 → P1 → P2 → P1  
**Gantt**: 

#### 4. ****Prioridades****:

* Cada proceso tiene una prioridad asignada; se ejecuta el de mayor prioridad.

**Ejemplo**: **Orden de ejecución**: P2 → P3 → P1  
**Gantt**: 

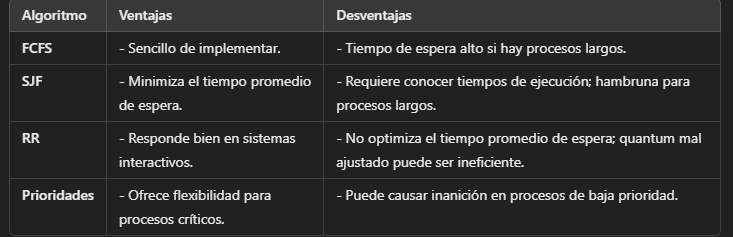
**(b) Parámetros necesarios**

* **FCFS**: No requiere parámetros adicionales. Solo necesita el orden de llegada.
* **SJF**: Requiere conocer los tiempos de ejecución de los procesos.
* **Round Robin**: Requiere un parámetro adicional: el **quantum** (tiempo fijo asignado a cada proceso).
* **Prioridades**: Requiere un valor de prioridad asociado a cada proceso.

**(c) Algoritmo más adecuado según tipo de procesos y/o SO**

1. **FCFS**:
   * **Adecuado para**: Sistemas simples o cuando los tiempos de ejecución son similares.
   * **No adecuado para**: Sistemas interactivos o tiempo compartido.
2. **SJF**:
   * **Adecuado para**: Procesos por lotes con tiempos de ejecución conocidos.
   * **No adecuado para**: Procesos interactivos; introduce problemas de hambruna para procesos largos.
3. **Round Robin**:
   * **Adecuado para**: Sistemas de tiempo compartido e interactivos.
   * **No adecuado para**: Procesos con alta carga de CPU o cuando el quantum es mal configurado.
4. **Prioridades**:
   * **Adecuado para**: Sistemas con diferentes niveles de importancia entre procesos.
5. **No adecuado para**: Sistemas donde todos los procesos tienen prioridad similar; puede provocar inanición para procesos con baja prioridad.

**(d) Ventajas y desventajas**

****

**4**)

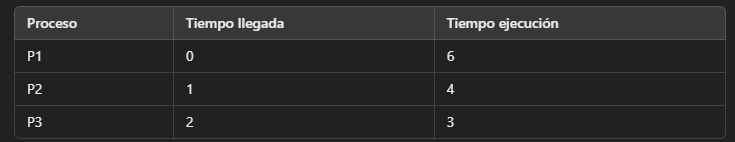
. Para el algoritmo Round Robin, existen 2 variantes: Timer Fijo Timer Variable

**(a) ¿Qué significan las variantes "Timer Fijo" y "Timer Variable"?**

1. **Timer Fijo**:
   * Todos los procesos reciben el mismo tiempo de ejecución (quantum) para cada turno.
   * Es una implementación uniforme que no discrimina entre procesos.
2. **Timer Variable**:
   * El tiempo de ejecución asignado (quantum) puede variar entre procesos.
   * Puede depender de factores como la prioridad, el comportamiento reciente del proceso, o características dinámicas del sistema.

**(b) Diferencias mediante un ejemplo**

**Supongamos 3 procesos con los siguientes tiempos de llegada y tiempos de ejecución**:

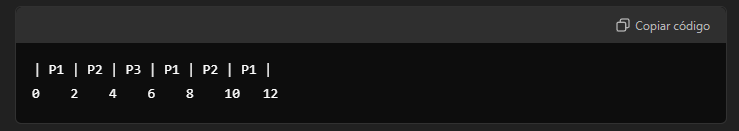
****

#### 1. ****Timer Fijo**** (Quantum = 2)

* Todos los procesos obtienen 2 unidades de tiempo por turno.

**Orden de ejecución**:  
P1 (2) → P2 (2) → P3 (2) → P1 (2) → P2 (2) → P1 (2).

**Gantt**:

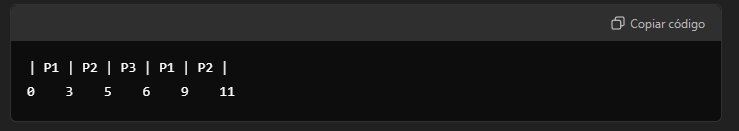
****

#### 2. ****Timer Variable****

* El quantum varía dependiendo del proceso. Por ejemplo:
  + P1: Quantum = 3
  + P2: Quantum = 2
  + P3: Quantum = 1

**Orden de ejecución**:  
P1 (3) → P2 (2) → P3 (1) → P1 (3) → P2 (2).

**Gantt**:

****

**(c) Dónde debería residir la información del Quantum**

1. **Timer Fijo**:
   * El quantum puede residir en una **estructura global del sistema operativo**.
   * Ejemplo: Una variable global compartida por el *scheduler*.
2. **Timer Variable**:
   * El quantum debe ser almacenado en una **estructura asociada al proceso** (e.g., en su PCB - *Process Control Block*).
   * Esto permite que cada proceso tenga un quantum único que el *scheduler* puede leer y utilizar dinámicamente.

### (a) Diagrama de Gantt: Planificación por Prioridades

Supongamos los siguientes procesos, donde **menor prioridad numérica indica mayor prioridad**:

| **Proceso** | **Tiempo llegada** | **Tiempo ejecución** | **Prioridad** |
| --- | --- | --- | --- |
| P1 | 0 | 5 | 3 |
| P2 | 1 | 3 | 1 |
| P3 | 2 | 2 | 2 |
| P4 | 3 | 4 | 4 |

#### i. ****No Apropiativa****

* Una vez que un proceso comienza, no puede ser interrumpido, incluso si llega otro proceso con mayor prioridad.

**Orden de ejecución**:  
P1 → P2 → P3 → P4  
**Gantt**:

Copiar código

| P1 | P2 | P3 | P4 |

0 5 8 10 14

#### ii. ****Apropiativa****

* Si llega un proceso con mayor prioridad, el proceso actual es interrumpido y se ejecuta el de mayor prioridad.

**Orden de ejecución**:  
P1 → P2 → P3 → P1 → P4  
**Gantt**:

Copiar código

| P1 | P2 | P3 | P1 | P4 |

0 1 4 6 8 12

### (b) Cálculo de TR (Tiempo de Retorno) y TE (Tiempo de Espera)

**Fórmulas**:

* **TR = Tiempo final - Tiempo de llegada**
* **TE = TR - Tiempo de ejecución**

#### No Apropiativa

| **Proceso** | **Tiempo llegada** | **Tiempo ejecución** | **Tiempo final** | **TR** | **TE** |
| --- | --- | --- | --- | --- | --- |
| P1 | 0 | 5 | 5 | 5 | 0 |
| P2 | 1 | 3 | 8 | 7 | 4 |
| P3 | 2 | 2 | 10 | 8 | 6 |
| P4 | 3 | 4 | 14 | 11 | 7 |

**TPR = (5 + 7 + 8 + 11) / 4 = 7.75**  
**TPE = (0 + 4 + 6 + 7) / 4 = 4.25**

#### Apropiativa

| **Proceso** | **Tiempo llegada** | **Tiempo ejecución** | **Tiempo final** | **TR** | **TE** |
| --- | --- | --- | --- | --- | --- |
| P1 | 0 | 5 | 8 | 8 | 3 |
| P2 | 1 | 3 | 4 | 3 | 0 |
| P3 | 2 | 2 | 6 | 4 | 2 |
| P4 | 3 | 4 | 12 | 9 | 5 |

**TPR = (8 + 3 + 4 + 9) / 4 = 6**  
**TPE = (3 + 0 + 2 + 5) / 4 = 2.5**

### (8) Ventajas y desventajas del algoritmo de prioridades

#### Ventajas:

1. **Flexibilidad**: Permite atender procesos críticos o urgentes con mayor rapidez.
2. **Optimización**: Ideal para sistemas donde algunos procesos tienen más impacto (e.g., tiempo real).
3. **Versatilidad**: Puede combinarse con otros algoritmos para mayor eficiencia.

#### Desventajas:

1. **Hambruna**: Los procesos con baja prioridad pueden quedarse sin ejecutarse.
2. **Complejidad**: Requiere una estrategia para asignar prioridades.
3. **Injusticia percibida**: Procesos "menos importantes" pueden ser postergados indefinidamente.

#### Uso según circunstancias:

1. **Adecuado para**:
   * Sistemas de tiempo real donde ciertos procesos son críticos.
   * Escenarios donde la prioridad puede determinarse claramente.
2. **No relevante para**:
   * Sistemas interactivos, donde el tiempo de respuesta uniforme es más importante.
   * Sistemas donde todos los procesos tienen igual importancia.

### Algoritmo SRTF (Shortest Remaining Time First)

#### (b) Ventajas frente a otros algoritmos

**Ventajas de SRTF**:

1. **Menor Tiempo Promedio de Retorno (TPR)**:
   * SRTF minimiza el tiempo promedio de retorno, siendo el más eficiente en sistemas donde este parámetro es crítico.
2. **Adecuado para sistemas con procesos cortos frecuentes**:
   * Ideal para escenarios donde llegan muchos procesos pequeños que necesitan ser atendidos rápidamente.
3. **Interrupciones dinámicas**:
   * Prioriza nuevos procesos con menor tiempo de ejecución restante, aumentando la capacidad de respuesta.

**Desventajas de SRTF**:

1. **Alta complejidad**:
   * Requiere recalcular el tiempo restante constantemente y realizar interrupciones frecuentes.
2. **Hambruna**:
   * Procesos con tiempos largos pueden sufrir retrasos si constantemente llegan procesos más cortos.

**Comparación con otros algoritmos**:

* **Mejor que FCFS**: Reduce significativamente el tiempo de espera y de retorno en sistemas con carga variada.
* **Mejor que SJF**: Permite atender procesos cortos más rápidamente sin esperar a que otros concluyan.
* **No adecuado para procesos críticos**: En sistemas donde ciertos procesos tienen alta prioridad, SRTF puede no ser la mejor opción.

En el algoritmo **Round Robin (RR)**, el **quantum** (o **cuanto de tiempo**) es el tiempo máximo que un proceso puede ejecutarse antes de que el sistema operativo lo interrumpa y pase a otro proceso de la cola de ejecución. El valor del quantum tiene un impacto significativo en el rendimiento del sistema, y la elección de un valor alto o bajo dependerá de la carga de trabajo y de los objetivos específicos del sistema.

**Casos para utilizar un quantum alto:**

1. **Procesos de ejecución larga o interactivos**:
   * Si los procesos suelen tener una duración prolongada o están diseñados para ejecutar tareas grandes y complejas, utilizar un valor alto de quantum puede ser beneficioso. Esto permite que un proceso se ejecute sin interrupciones frecuentes, reduciendo el overhead causado por los cambios de contexto (la sobrecarga que implica cambiar de un proceso a otro).
   * Ejemplo: En un sistema con aplicaciones que requieren procesamiento intensivo, como editores de video o simulaciones científicas, un quantum alto puede evitar que el proceso sea interrumpido constantemente.
2. **Baja sobrecarga de cambio de contexto**:
   * Si el sistema tiene pocos procesos o la sobrecarga de cambiar entre procesos es significativa, un quantum más alto puede ser útil, ya que reduce la cantidad de cambios de contexto y la consiguiente pérdida de rendimiento.
3. **Sistemas con pocos procesos**:
   * En entornos donde hay pocos procesos en ejecución, un quantum más largo permite que cada proceso tenga más tiempo para ejecutarse antes de ser interrumpido, sin generar una distribución de tiempo muy fragmentada entre los procesos.

**Ventajas de un quantum alto:**

1. **Reducción de la sobrecarga por cambios de contexto**:
   * Un quantum alto permite que cada proceso tenga más tiempo de ejecución antes de ser interrumpido, lo que reduce el número de cambios de contexto (transiciones entre procesos). Esto es especialmente beneficioso cuando los procesos son largos o requieren tiempos de procesamiento considerables.
2. **Mayor eficiencia para procesos largos**:
   * Los procesos que requieren una cantidad significativa de tiempo de CPU (como tareas de procesamiento intensivo o aplicaciones interactivas) se benefician de no ser interrumpidos frecuentemente, lo que mejora la eficiencia general.
3. **Mejor uso del CPU**:
   * Los procesos con una carga alta de CPU pueden aprovechar mejor el tiempo del procesador, ya que tienen la oportunidad de avanzar más en su ejecución antes de que se les ceda el control al siguiente proceso.

**Desventajas de un quantum alto:**

1. **Menor equidad entre procesos**:
   * Si el quantum es demasiado alto, los procesos más cortos o interactivos pueden tener que esperar mucho tiempo para acceder al CPU. Esto puede hacer que el sistema sea menos equitativo, especialmente en sistemas con muchos procesos de corta duración.
   * Ejemplo: En un sistema con muchos procesos de entrada/salida que son interactivos (como en servidores web), un quantum alto puede hacer que estos procesos tengan que esperar demasiado para obtener acceso al CPU.
2. **Peor respuesta para procesos interactivos**:
   * Los procesos interactivos (por ejemplo, los que involucran interfaces gráficas de usuario) se pueden ver perjudicados, ya que un quantum largo puede hacer que el sistema tarde mucho en responder a las solicitudes del usuario. Esto puede llevar a una experiencia de usuario deficiente en sistemas con interfaces de usuario.
3. **Posible desperdicio de CPU**:
   * Si el quantum es alto y un proceso termina antes de que se agote el tiempo asignado, el CPU podría quedarse ocioso, lo que disminuye la eficiencia del sistema al no aprovechar todo el tiempo disponible para otros procesos.

**Conclusión**

Un **quantum alto** puede ser útil en sistemas donde predominan procesos largos y CPU-intensivos, reduciendo la sobrecarga de cambios de contexto y mejorando la eficiencia en general. Sin embargo, en sistemas con muchos procesos pequeños o interactivos, un quantum demasiado alto puede generar un mal rendimiento y una experiencia de usuario deficiente debido a la mayor latencia y menor equidad entre los procesos. Por lo tanto, el valor del quantum debe ajustarse según las características y necesidades del sistema y los tipos de procesos que se ejecutan.

**(9) ¿Qué significa Inanición (Starvation)?**

La **inanición** o **starvation** en el contexto de la programación de procesos se refiere a una situación en la que un proceso no recibe la cantidad de recursos necesaria (en este caso, el tiempo de CPU) para ejecutarse durante un largo período de tiempo, o incluso nunca llega a ejecutarse. Esto generalmente ocurre cuando los recursos del sistema (como el procesador) se asignan constantemente a otros procesos, mientras que el proceso en cuestión está bloqueado o relegado al final de la cola de ejecución.

La inanición puede ocurrir si el proceso no cumple con las condiciones o prioridades necesarias para obtener acceso a los recursos y, como resultado, permanece "hambriento" de ejecución.

**(b) ¿Cuál/es de los algoritmos vistos puede provocarla?**

Varios algoritmos de planificación pueden provocar inanición, especialmente aquellos que dependen de la prioridad o de la duración de los trabajos. Algunos de los algoritmos que pueden dar lugar a inanición incluyen:

1. **Algoritmo de Prioridades (Non-preemptive y Preemptive)**:
   * En algoritmos de planificación por prioridades, un proceso de baja prioridad puede no recibir nunca el CPU si siempre hay procesos con una prioridad más alta listos para ejecutarse. Esto puede ocurrir especialmente en sistemas donde los procesos de alta prioridad son constantemente introducidos en la cola de ejecución, dejando a los procesos de baja prioridad esperando indefinidamente.
2. **SJF (Shortest Job First) y SRTF (Shortest Remaining Time First)**:
   * Estos algoritmos asignan el CPU al proceso con el menor tiempo de ejecución restante. Los procesos con tiempos de ejecución más largos o con tiempos de ejecución muy variables pueden ser interrumpidos constantemente por procesos más cortos, lo que puede provocar que nunca obtengan tiempo de CPU, creando un escenario de inanición.
3. **Round Robin**:
   * Aunque el algoritmo Round Robin es diseñado para evitar la inanición, en situaciones de alta carga con procesos de distinta prioridad, los procesos con mayor tiempo de ejecución pueden ser desplazados constantemente si tienen un tiempo de quantum bajo y no tienen una oportunidad de completar su ejecución. Sin embargo, la inanición en Round Robin es menos probable que en otros algoritmos, especialmente si se ajusta adecuadamente el quantum.

**(c) ¿Existe alguna técnica que evite la inanición para el/los algoritmos mencionados en b?**

Sí, existen varias técnicas y enfoques para evitar o mitigar la inanición en los algoritmos que podrían provocarla:

1. **Algoritmo de Prioridades**:
   * **Envejecimiento (Aging)**: Una técnica común para evitar la inanición en algoritmos de prioridad es el envejecimiento, en la que la prioridad de un proceso aumenta gradualmente mientras espera en la cola de procesos. Esto asegura que, eventualmente, los procesos de baja prioridad reciban tiempo de CPU. El envejecimiento evita que un proceso quede atrapado esperando indefinidamente si hay otros procesos con prioridad más alta.
2. **SJF y SRTF**:
   * **Ajuste de Prioridades o Envejecimiento (Aging)**: Similar a los algoritmos de prioridad, se puede aplicar envejecimiento para evitar la inanición en SJF y SRTF. Los procesos que quedan atrapados debido a que tienen una duración de ejecución más larga pueden ver su "tiempo restante" disminuir gradualmente para que eventualmente puedan ser ejecutados.
   * **Preemptive SJF**: En el caso de SRTF, aunque los procesos de duración más corta tienen prioridad, una vez que los procesos más largos tienen tiempo suficiente para ejecutarse, es menos probable que sufran inanición si se les da oportunidad después de que se completen los trabajos más cortos.
3. **Round Robin**:
   * Aunque Round Robin es menos susceptible a la inanición debido a su naturaleza equitativa, se puede evitar aún más la inanición ajustando adecuadamente el **quantum**. Si el quantum es demasiado corto, un proceso de larga duración podría nunca obtener suficiente tiempo para ejecutarse. Aumentar el tamaño del quantum para ciertos tipos de procesos (o priorizar procesos que llevan más tiempo) podría ayudar a evitar la inanición.
4. **Implementación de reglas de prioridades combinadas**:
   * Algunos sistemas operativos implementan combinaciones de algoritmos para asegurar que se atienda a los procesos más urgentes sin descuidar a los más largos. Por ejemplo, combinando Round Robin con SJF o utilizando un sistema de prioridades con envejecimiento.

**Conclusión:**

La **inanición** es un problema potencial en varios algoritmos de planificación, particularmente aquellos basados en prioridades o en el tiempo de ejecución estimado de los procesos. Las técnicas como el **envejecimiento** o el **ajuste dinámico de prioridades** son comúnmente utilizadas para prevenir la inanición y asegurar que todos los procesos tengan la oportunidad de ejecutarse eventualmente.

**11)**

**(a) Round Robin**

El algoritmo **Round Robin** es equitativo, ya que asigna un quantum de tiempo fijo a cada proceso en un ciclo continuo. Sin embargo, presenta desventajas cuando hay una combinación de procesos **ligados a la CPU** y **ligados a E/S**:

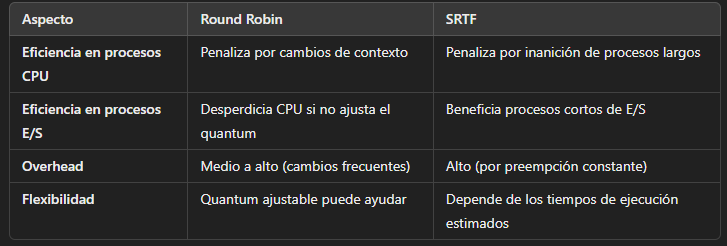
1. **Desventajas para procesos ligados a la CPU:**
   * Los procesos ligados a la CPU, que necesitan largos periodos de ejecución ininterrumpida, se ven penalizados al ser constantemente interrumpidos por el cambio de contexto. Esto genera una sobrecarga adicional debido al **overhead del cambio de contexto**, lo que disminuye la eficiencia del sistema.
2. **Desventajas para procesos ligados a E/S:**
   * Estos procesos suelen pasar mucho tiempo en espera de operaciones de entrada/salida. Debido al quantum fijo, podrían desperdiciar tiempo de CPU asignado porque no están listos para ejecutarse, lo que disminuye la utilización efectiva del procesador.
3. **Implicaciones generales:**
   * El sistema puede parecer equitativo, pero si el quantum no se ajusta adecuadamente, podría beneficiar más a procesos cortos o ligados a E/S, mientras penaliza a procesos largos o ligados a la CPU.

**(b) SRTF (Shortest Remaining Time First)**

El algoritmo **SRTF** prioriza el proceso con el menor tiempo de ejecución restante, lo que lo hace ideal para minimizar el tiempo promedio de retorno. Sin embargo, tiene desventajas importantes en un entorno mixto de procesos ligados a CPU y E/S:

1. **Desventajas para procesos ligados a la CPU:**
   * Los procesos con tiempos largos de ejecución son constantemente interrumpidos si llegan nuevos procesos con tiempos más cortos. Esto puede llevar a una forma de **inanición**, donde los procesos ligados a la CPU quedan relegados indefinidamente si los procesos cortos son frecuentes.
2. **Desventajas para procesos ligados a E/S:**
   * Los procesos de E/S suelen tener ráfagas cortas de CPU, seguidas de periodos de espera en dispositivos de entrada/salida. Aunque estos procesos se priorizan debido a sus ráfagas cortas, el sistema podría generar demasiadas interrupciones y cambios de contexto, aumentando el **overhead**.
3. **Implicaciones generales:**
   * Si el sistema tiene un gran número de procesos cortos, los procesos más largos quedan desatendidos. Esto puede ser problemático en sistemas donde los procesos largos son críticos para el rendimiento general.

**Comparación y conclusiones:**

**Conclusión:**

* **Round Robin** es mejor para sistemas con una mezcla uniforme de procesos si el quantum está bien ajustado, pero no es ideal para procesos intensivos en CPU.
* **SRTF** es eficiente en sistemas con muchos procesos cortos, pero debe usarse con cuidado para evitar inanición y sobrecarga excesiva en sistemas mixtos.+

**13)**

En un algoritmo de **VRR (Virtual Round Robin)** con **Timer Variable**, puede suceder que el quantum de un proceso **nunca llegue a 0**, dependiendo de cómo se gestionen los procesos y las interrupciones.

**Justificación:**

1. **Proceso interrumpido antes de agotar su quantum:**  
   Si un proceso es preemptado antes de consumir todo su quantum (por ejemplo, debido a una operación de E/S o por la llegada de un proceso de mayor prioridad), el quantum restante se guarda. Cuando este proceso regresa a la cola auxiliar, se le asigna nuevamente el tiempo restante, pero si vuelve a ser interrumpido de manera similar, su quantum podría nunca llegar a 0.
2. **Cola auxiliar con prioridad:**  
   En VRR, los procesos en la cola auxiliar tienen prioridad. Si un proceso en la cola principal es repetidamente desplazado por procesos que ingresan y salen rápidamente de la cola auxiliar, puede no llegar a ejecutar lo suficiente como para consumir todo su quantum.
3. **Desbalance en interrupciones:**  
   Si el sistema experimenta interrupciones frecuentes (por ejemplo, llegada de E/S, señales externas o cambios de contexto), el quantum de un proceso podría no decrementarse completamente antes de ser pausado. Esto podría perpetuar un ciclo en el cual el proceso nunca alcanza 0.

**Conclusión:**

Aunque poco común, este comportamiento es posible en sistemas con VRR y Timer Variable si el manejo de interrupciones o prioridades no está optimizado. Un diseño robusto debe considerar este escenario para evitar que procesos específicos queden atrapados sin completar su ejecución.

### 15)(a) Algoritmo para administrar las colas de procesos interactivos y batch

1. **Cola de procesos interactivos:**  
   Estos procesos tienen tiempos de respuesta críticos y requieren atención rápida.
   * **Algoritmo sugerido:** **Round Robin (RR)**
     + Permite un reparto justo y mantiene bajos los tiempos de respuesta.
     + Se puede ajustar el quantum para equilibrar rapidez y eficiencia.
2. **Cola de procesos batch:**  
   Estos procesos son menos sensibles al tiempo de respuesta, pero pueden ser largos y consumir muchos recursos.
   * **Algoritmo sugerido:** **Shortest Job First (SJF) o SRTF (Shortest Remaining Time First)**
     + Priorizan procesos cortos, optimizando el tiempo promedio de retorno (TR).
     + En caso de alta variabilidad, SRTF puede ser más eficiente al reordenar dinámicamente.

**(b) Algoritmo para planificar entre las colas de procesos interactivos y batch**

Para elegir qué cola priorizar, el algoritmo debe garantizar que los procesos interactivos tengan preferencia, pero sin bloquear completamente a los batch.

* **Algoritmo sugerido:** **Planificación por prioridades con envejecimiento**
  + Los procesos de la cola interactiva tienen mayor prioridad.
  + A medida que los procesos batch esperan, su prioridad aumenta gradualmente, evitando inanición (starvation).

**Justificación de los algoritmos:**

* **Round Robin para procesos interactivos**: Asegura una respuesta rápida y equilibrada, fundamental para tareas críticas.
* **SJF/SRTF para procesos batch**: Optimiza el tiempo de procesamiento general al manejar eficientemente las tareas largas.
* **Planificación por prioridades entre colas**: Combina rapidez en procesos críticos con eficiencia global al atender procesos menos urgentes sin retrasos excesivos.

Si el **quantum (q)** en el algoritmo **Round-Robin (RR)** se incrementa sin límite, el algoritmo comienza a comportarse de manera similar al **FIFO (First In, First Out)** por las siguientes razones:

**Funcionamiento de Round-Robin con quantum grande:**

1. **Round-Robin y el quantum:**  
   En RR, cada proceso se ejecuta durante un tiempo máximo igual al quantum antes de ser interrumpido y enviado al final de la cola.
2. **Quantum muy grande:**  
   Si el valor de qqq es mucho mayor que las ráfagas de CPU de los procesos, cada proceso podrá completar su ejecución completamente antes de que el quantum expire. Esto significa que un proceso no será interrumpido, sino que continuará ejecutándose hasta terminar.

**Similitud con FIFO:**

* En **FIFO**, los procesos se ejecutan en el orden en que llegaron a la cola, y cada proceso completa su ejecución antes de pasar al siguiente.
* En **RR** con qqq extremadamente grande, el resultado será el mismo: cada proceso terminará completamente en el orden de llegada, ya que el quantum no será un límite práctico.

**Conclusión:**

Cuando el quantum es muy grande en Round-Robin:

* **El preemptive nature** del algoritmo se pierde, ya que los procesos no son interrumpidos.
* El algoritmo RR se comporta como FIFO, ya que los procesos se ejecutan completamente en el orden en el que llegaron.

Esto demuestra que el quantum es un parámetro clave para diferenciar RR de otros algoritmos de planificación como FIFO.

### 22)(a) ¿Con cuál/es de estas clasificaciones se asocian las PCs de escritorio habituales?

1. **Homogéneos:**
   * Las PCs de escritorio suelen tener procesadores homogéneos, ya que todos los núcleos dentro del CPU (en arquitecturas modernas) son iguales y comparten el mismo conjunto de instrucciones.
2. **Multiprocesador fuertemente acoplado:**
   * Los núcleos de un CPU comparten una memoria principal y trabajan bajo el control de un único sistema operativo.
3. **Procesadores especializados:**
   * Las PCs modernas también pueden incluir procesadores especializados, como GPUs (procesadores gráficos), controladores de E/S, o procesadores para tareas específicas como aceleración criptográfica.

**(b) ¿Qué significa que la asignación de procesos se realice de manera simétrica?**

La **asignación simétrica de procesos** (Symmetric Multiprocessing o SMP) significa que:

* **Todos los procesadores tienen igual acceso** a la memoria, dispositivos y al sistema operativo.
* El sistema operativo puede asignar procesos o hilos a cualquier procesador de forma equitativa.
* **Ventajas:**
  + Balance de carga entre procesadores.
  + Mayor tolerancia a fallos, ya que no hay un procesador único en control.

**(c) ¿Qué significa que se trabaje bajo un esquema Maestro/Esclavo?**

El esquema **Maestro/Esclavo** (Master/Slave Multiprocessing) implica que:

* **Un procesador principal (Maestro):**
  + Es responsable de las decisiones de planificación, asignación de tareas y coordinación general del sistema.
* **Procesadores secundarios (Esclavos):**
  + Ejecutan las tareas asignadas por el Maestro, pero no tienen control sobre la planificación o el sistema operativo.

**Ventajas:**

* Simplicidad en la implementación, ya que el Maestro centraliza la lógica de control.

**Desventajas:**

* El Maestro puede convertirse en un cuello de botella si está sobrecargado.
* Menor tolerancia a fallos, ya que si el Maestro falla, todo el sistema se detiene.

### ****(a) ¿Cuál sería el método de planificación más sencillo para asignar CPUs a los procesos?****

En el caso de procesadores homogéneos, el método más sencillo sería **Cola Única con Asignación Equitativa (First-Come-First-Served, FCFS)** o una **Distribución por Round Robin (RR)**. Ambos métodos son simples y eficientes para sistemas homogéneos:

1. **FCFS:**
   * Los procesos son atendidos en el orden en que llegan.
   * No hay preemption; un proceso ocupa el CPU hasta que termine su ejecución.
2. **Round Robin:**
   * Cada proceso recibe un tiempo fijo de ejecución (quantum).
   * Si un proceso no termina en ese tiempo, se coloca nuevamente al final de la cola.

### ****(b) Ventajas y desventajas del método escogido****

#### ****FCFS:****

**Ventajas:**

* Simple de implementar.
* No requiere context switches frecuentes, por lo que es eficiente en cuanto al uso de CPU.

**Desventajas:**

* **Problema de Convoy Effect:** Procesos cortos pueden quedar bloqueados detrás de procesos largos.
* Ineficiente para sistemas interactivos, ya que no prioriza procesos de respuesta rápida.

#### ****Round Robin:****

**Ventajas:**

* Garantiza **equidad**, ya que todos los procesos tienen acceso al CPU por igual.
* Es adecuado para sistemas interactivos, ya que los procesos tienen tiempos de respuesta rápidos.

**Desventajas:**

* Depende del **valor del quantum:**
  + Si es demasiado pequeño, aumenta el número de context switches, afectando el rendimiento.
  + Si es demasiado grande, puede parecerse a FCFS.
* No considera la naturaleza del proceso (CPU-bound o I/O-bound), lo que puede llevar a una asignación ineficiente del CPU.

En general, para un entorno homogéneo donde la equidad es esencial, **Round Robin** es más adecuado, pero el valor del quantum debe ajustarse cuidadosamente para equilibrar el rendimiento y la equidad.

### 23(a) Huella de un proceso en un procesador

La "huella" de un proceso en un procesador se refiere a los datos específicos de ese proceso almacenados en la caché del procesador y en otros componentes internos, como registros y memoria local. Estos datos incluyen instrucciones ejecutadas recientemente, variables y datos intermedios utilizados en la ejecución del proceso.

**(b) Afinidad con un procesador**

La afinidad se refiere a la asociación preferencial de un proceso con un procesador específico en un sistema multiprocesador. Esto significa que un proceso tiene prioridad para ejecutarse en el mismo procesador en el que se ejecutó anteriormente, para maximizar el uso de la memoria caché y reducir los costos de intercambio de datos.

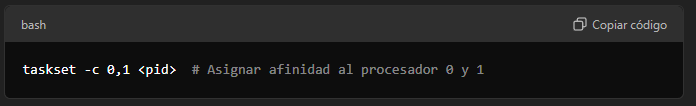
**(c) ¿Por qué podría ser mejor en algunos casos que un proceso se ejecute en el mismo procesador?**

Ejecutar un proceso en el mismo procesador mejora el rendimiento debido a:

1. **Reutilización de la caché:** Los datos e instrucciones utilizados anteriormente permanecen en la caché del procesador, lo que reduce los accesos a la memoria principal.
2. **Minimización de overhead:** Evita los costos asociados con la migración de procesos entre procesadores, como la sincronización y transferencia de datos.

**(d) ¿Puede el usuario en Windows cambiar la afinidad de un proceso? ¿Y en GNU/Linux?**

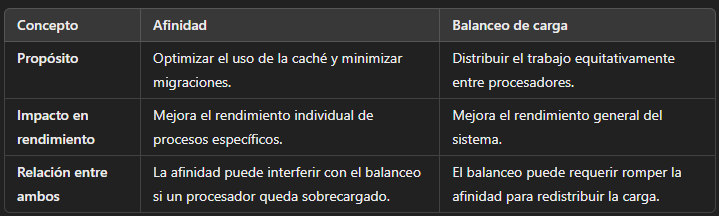
1. **Windows:** Sí, los usuarios pueden cambiar la afinidad de un proceso desde el **Administrador de Tareas**. Se selecciona el proceso, se hace clic derecho, y se elige la opción "Establecer afinidad".
2. **GNU/Linux:** Sí, el usuario puede configurar la afinidad utilizando comandos como taskset. Ejemplo:

****

**(e) Concepto de balanceo de carga (load balancing)**

El balanceo de carga es una técnica utilizada para distribuir la carga de trabajo de manera equitativa entre los procesadores de un sistema multiprocesador. Su objetivo principal es evitar que algunos procesadores estén sobrecargados mientras otros permanecen inactivos, garantizando un uso eficiente de los recursos y mejorando el rendimiento general.

**(f) Comparación entre afinidad y balanceo de carga**

**Conclusión:**  
El uso simultáneo de afinidad y balanceo requiere equilibrio:

* En sistemas con tareas críticas de rendimiento, la afinidad puede ser prioritaria.
* En sistemas con alta carga y múltiples usuarios, el balanceo puede ser más relevante.