1) Responde en forma sintética sobre los siguientes conceptos:

a) Programa y proceso.

Programa: Conjunto de instrucciones escritas en un lenguaje de programaciṕn que una computadora puede ejecutar para realizar una tarea específica.

Proceso: Instancia de un programa en ejecución, que incluye el código de ejecución, los recursos asignados, y el estado de su ejecuciṕon en un momento dado.

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

Retorno: Tiempo que transcurre entre que el proceso llega al sistema hasta que completa su ejecución (Fin - Inicio + 1).

Espera: Tiempo que el proceso se encuentra en el sistema esperando, es decir el tiempo que pasa sin ejecutarse (TR - Tiempo CPU).

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

Los tiempos promedios se calculan sumando todos los tiempos y dividiendo por la cantidad de procesos usados para los cálculos.

d) ¿Qué es el Quantum?

Medida que determina cuánto tiempo podrá usar el procesos para cada proceso:

Quantum pequeño:

- Mayor cantidad de respuestas.
- Equidad mejorada.
- Mayor sobrecarga de cambio de contexto.
- Menor eficiencia para procesos largos.

Quantum grande:

- Menor sobrecarga de cambio de contexto.
- Apropiado para procesos largos.
- Menor capacidad de respuesta.
- Posible falta de equidad.

e) ¿Qué significa que un algoritmo de scheduling sea apropiativo y no apropiativo? (Preemptive o Non-Preemptive)?

Preemptive: El proceso en ejecución puede ser interrumpido y llevado a la cola de listos:

- Mayor overhead pero mejor servicio.
- Un proceso no monopoliza el procesador.

Overhead: Recursos adicionales requeridos para realizar una tarea o administrar un sistema.

Non-preemptive: Una vez que un proceso está en estado de ejecución, continúa hasta que termina o se bloquea por algún evento (ej: I/O).

f) ¿Qué tareas realizan?

Short Term Scheduler: Determina qué proceso pasará a ejecutarse.

Long Term Scheduler: Admite nuevos procesos a memoria (controla el grado de multiprogramación).

Medium Term Scheduler: Realiza el swapping (intercambio) entre el disco y la memoria cuando el SO lo determina (puede disminuir el grado de multiprogramación).

g) ¿Qué tareas realiza el Dispatcher?

El Dispatcher es una parte fundamental del sistema operativo que se encarga de gestionar la ejecución de los procesos. Sus tareas principales son:

- Selección del proceso listo: El Dispatcher selecciona un proceso de la cola de procesos listos para ejecutarse, eligiendo cuál de ellos debe pasar a la CPU.
- Context switch: Cuando se selecciona un nuevo proceso, el dispatcher realiza un cambio de contexto, que consiste en guardar el estado (registro, contador de programa, etc) del proceso que estaba en ejecución y cargar el estado del proceso que va a comenzar.
- Transferencia de control: Una vez realizado el cambio de contexto, el Dispatcher transfiere el control de la CPU al proceso seleccionado, permitiéndo continuar su ejecución.
- Gestión de la planificación de procesos: Aunque el Dispatcher no decide qué proceso ejecutar, trabaja en conjunto con el planificador (scheduler) para ejecutar el proceso que ha sido seleccionado por este último.

2) Procesos:

i) Investigue y detalle para qué sirve cada uno de los siguientes comandos. (Puede que algún comando no venga por defecto en su distribución, por lo que deberá instalarlo).

Comando	Descripción	Uso principal	Comentarios adicionales
`top`	Muestra una vista en tiempo real de los procesos en ejecución.	Monitorizar la actividad de la CPU, memoria y procesos.	Viene instalado por defecto en la mayoría de las distribuciones Linux.
`htop`	Herramienta interactiva como `top`, pero más amigable y con interfaz mejorada.	Similar a `top`, pero permite usar teclas para matar procesos o renombrar prioridades.	Necesita instalación en algunas distribuciones. Ejemplo: `sudo apt install htop`.
`ps`	Muestra una lista estática de procesos en ejecución.	Examinar procesos y su información detallada.	Se usa con opciones como `ps aux` para más detalles.
`pstree`	Muestra los procesos en forma de árbol jerárquico.	Visualizar la relación padre-hijo entre procesos.	Útil para comprender cómo los procesos están relacionados entre sí.
`kill`	Envía señales a un proceso para realizar acciones, como terminarlo.	Finalizar o controlar un proceso específico.	Requiere el **PID** del proceso. Ejemplo: `kill -9 1234` para matar un proceso.
`pgrep`	Busca procesos por nombre y devuelve sus PIDs.	Localizar procesos sin necesidad de buscar manualmente en `ps`.	Ejemplo: `pgrep firefox`.
`killall`	Mata todos los procesos con un nombre específico.	Terminar múltiples procesos con un comando.	Ejemplo: `killall firefox`.
`renice`	Cambia la prioridad de un proceso en ejecución.	Ajustar la prioridad de un proceso para dar más o menos recursos.	Ejemplo: `renice +5 1234` (PID 1234 será menos prioritario).
`xkill`	Mata aplicaciones gráficas mediante un clic.	Forzar el cierre de ventanas en entornos gráficos.	Puede necesitar instalación: `sudo apt install x11-utils`.
`atop`	Monitoriza el sistema y registra métricas detalladas del uso de CPU, memoria, disco y red.	Análisis detallado del rendimiento y registro para revisiones futuras.	Similar a `top`, pero con un enfoque en análisis a largo plazo. Ejemplo: `sudo apt install atop`.

ii) Observe detenidamente el siguiente código. Intente entender lo que hace sin necesidad de ejecutarlo.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
        int c;
        pid_t pid;
        printf("Comienzo.:\n");
        for (c = 0; c < 3 ; c++ )
        {
                pid = fork();
        }
        printf("Proceso\n");
        return 0;
}
```

- **a)** ¿Cuántas líneas con la palabra “Proceso” aparecen al final de la ejecución de este programa?
- **b)** ¿El número de líneas es el número de procesos que han estado en ejecución? Ejecuta el programa y compruebe si su respuesta es correcta. Modifique el valor del bucle for y compruebe los nuevos resultados.

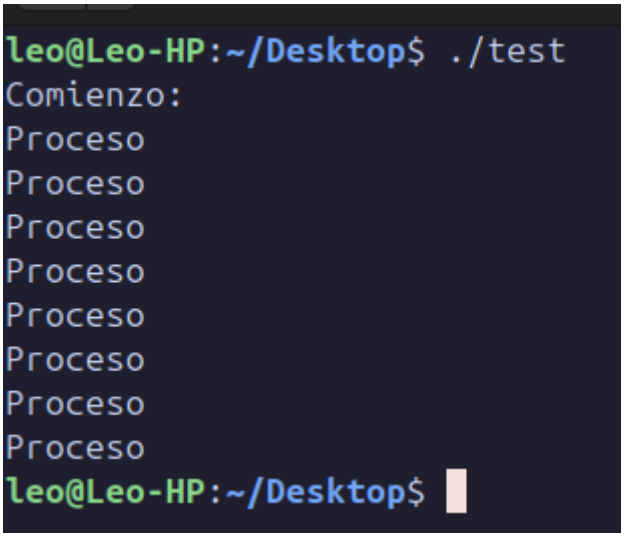
La llamada de **fork()** crea un nuevo proceso, duplicando el proceso padre. Cada vez que se llama, se generan 2 procesos (el padre y el hijo).
El bucle **for** se ejecuta 3 veces, por lo que cada iteración llama a **fork()**, lo que aumenta el número de procesos.

El número total de procesos creados se puede calcular como 2^n, donde n es el número de llamadas a **fork()** en la ejecución de un proceso padre.

En este caso, el bucle ejecuta **fork()** 3 veces:

- 2^3 = 8 procesos en total.

La salida de **printf(“Proceso\n”)** se mostrará en pantalla 8 veces.



iii) Vamos a tomar una variante del programa anterior. Ahora, además de un mensaje, vamos a añadir una variable y, al final del programa vamos a mostrar su valor. El nuevo código del programa se muestra a continuación.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
        int c;
        int p=0;
        pid_t pid;
        for (c = 0; c < 3 ; c++ )
        {
                pid = fork();
        }
        p++;
        printf("Proceso %d\n", p);
        return 0;
}
```

a) ¿Qué valores se muestran por consola?
b) ¿Todas las líneas tendrán el mismo valor o algunas líneas tendrán valores distintos?
c) ¿Cuál es el valor (o valores) que aparece?. Ejecuta el programa y compruebe si su respuesta es correcta. Modifique el valor del bucle for y el lugar donde se incrementa la variable **p** y compruebe los nuevos resultados.
d) ¿Qué tipo de comunicación es posible con pipes?

```
leo@Leo-HP:~/Desktop$ ./test
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
leo@Leo-HP:~/Desktop$
```

Cada proceso imprimirá su versión de la variable **p**, que siempre es 1 ya que se incrementa en cada proceso de manera independiente.

```
leo@Leo-HP:~/Desktop$ cat test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    int c;
    int p = 0;
    pid_t pid;

    for (c = 0; c < 3; c++) {
        pid = fork();
        p++;
    }

    printf("Proceso %d\n", p);

    return 0;
}
leo@Leo-HP:~/Desktop$ ./test
Proceso 3
Proceso 3
Proceso 3
Proceso 3
Proceso 3
Proceso 3
Proceso 3
Proceso 3
leo@Leo-HP:~/Desktop$
```

Mismo resultado si pongo **p++** antes de **fork()**

Cuando **p++** está antes del **fork()**, se incrementa en el proceso actual (el que ejecuta **fork()**) antes de que se cree el nuevo proceso.

- Los hijos heredan el valor actual de **p**, pero no vuelven a ejecutarse las iteraciones anteriores del bucle.
- Esto hace que el valor de **p** acumulado refleje la cantidad de iteraciones completadas por el proceso inicial antes de crear al hijo.
-

Cuando **p++** está después del **fork**(), cada proceso, ya sea el padre o el hijo incrementa su propia copia de **p** después de la bifurcación.

- Aunque se crean múltiples procesos, cada uno tiene su propia versión aislada de **p**, y esta se incrementa solo en función de las iteraciones que realiza dicho proceso.

Aunque hay 2^3 = 8 procesos creados, cada uno ejecuta exactamente tres incrementos de **p**, ya que el bucle se ejecuta completamente en cada proceso creado.
Por lo tanto, todos los procesos imprimen **p = 3**.

Los pipes permiten la comunicación unidireccional entre procesos.

- Los datos fluyen de un extremo (escritor) al otro (lector).
- Generalmente se usan entre un proceso padre e hijo o entre procesos relacionados.

Se utiliza el sistema de llamada **pipe()** para crear un canal de comunicación. Tiene dos extremos:

- Extremo de escritura: Los datos son enviados desde aquí.
- Extremo de lectura: Los datos son recibidos desde aquí.

iv) Comunicación entre procesos:

a) Investigue la forma de comunicación entre procesos a través de pipes.
b) ¿Cómo se crea un pipe en C?
c) ¿Qué parámetro es necesario para la creación de un pipe?. Ejemplifique para qué se utiliza.
d) ¿Qué tipo de comunicación es posible con pipes?

Los pipes son una forma de comunicación entre procesos en sistemas UNIX/Linux. Permiten transferir datos de un proceso a otro de manera unidireccional, mediante un buffer temporal gestionado por el kernel.

Un pipe se crea con la función **pipe()**, que toma un array de enteros como argumentos. Estos enteros representan:

- **fd[0]**: El extremo de lectura del pipe.
- **fd[1]**: El extremo de escritura del pipe.

```c
#include <unistd.h>
int pipe(int fd[2]);
```

La función devuelve:

- 0 si el pipe se creó exitosamente.
- -1 si ocurrió un error.

Comunicación unidireccional:

- Los datos fluyen en una sola dirección: desde el extremo de escritura (**fd[1]**) hasta el extremo de lectura (**fd[0]**).
- Para comunicación bidireccional, se necesitan dos pipes (uno para cada dirección).

Usos comunes:

- Transmitir datos entre un proceso padre y sus procesos hijos.
- Implementar patrones como productos-consumidor, donde un proceso genera datos y el otro los consume.

Restricciones:

- Los pipes solo funcionan entre procesos relacionados (como un padre y sus hijos).
- El tamaño del buffer del pipe es limitado (generalmente unos pocos kb).

v) ¿Cuál es la información mínima que el SO debe tener sobre un proceso? ¿En qué estructura de datos asociada almacena dicha información?

El sistema operativo debe mantener información sobre cada proceso en una estructura llamada **PCB** (Process Control Block). Esta información mínima incluye:

- Identificación del proceso: **PID**
- Estado del proceso: **Ejecución, bloqueado, listo**
- Contexto de la CPU**:** Valores de registros, incluyendo el **PC**, entre otros.
- Información de memoria: Punteros a las tablas de memoria que indican el espacio que ocupa el proceso.
- Información de control de E/S: Recursos de E/S que indican qué espacio ocupa el proceso.
- Información de prioridades: Nivel de prioridad del proceso para planificación.
- Recursos asignados: Cantidad de memoria, tiempo de CPU utilizado y límites.

vi) ¿Qué significa que un proceso sea “CPU Bound” y “I/O Bound”?

CPU Bound: Un proceso es **CPU Bound** si pasa la mayor parte de su tiempo ejecutando cálculos en el procesador.

Su rendimiento depende principalmente de la velocidad de la CPU.

Ejemplo: Procesos de simulación científica o criptografía.

I/O Bound: Un proceso es **I/O Bound** si pasa más tiempo esperando operaciones de E/S que ejecutándose en la CPU.

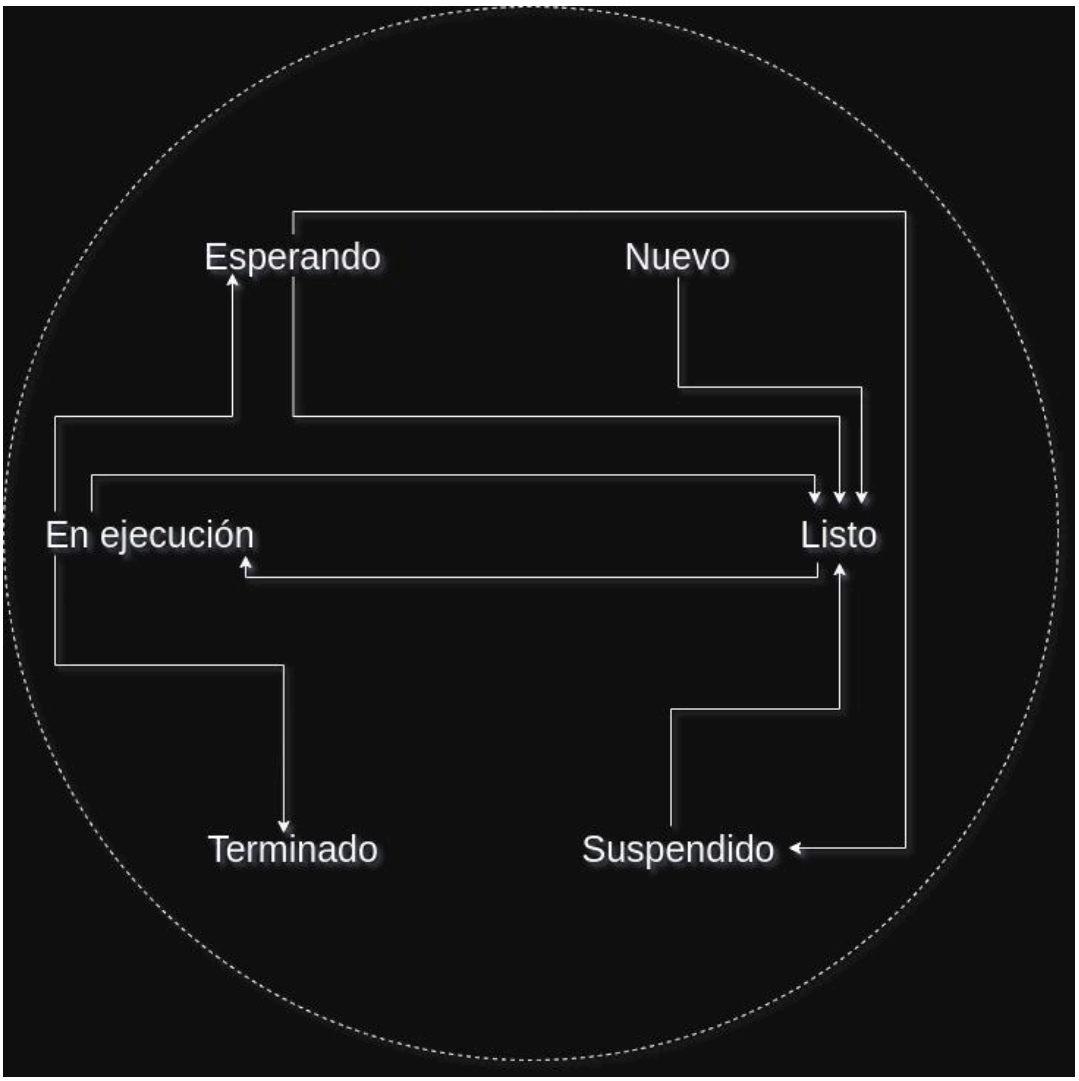
Su rendimiento depende de la velocidad de los dispositivos de E/S.

Ejemplo: Procesos de lectura/escritura de archivos o transferencia de datos.

vii) ¿Cuáles son los estados posibles por lo que puede atravesar un proceso?
viii) Explique mediante un diagrama las posibles transiciones entre los estados.

- Nuevo (new): El proceso está siendo creado.
- Listo (ready): El proceso está en cola, esperando que el planificador lo asigne a la CPU.
- En ejecución (running): El proceso está siendo ejecutado por la CPU.
- Bloqueado/Esperando (waiting): El proceso espera un evento externo (como completar una operación de E/S).
- Terminado (terminated): El proceso ha finalizado su ejecución.
- Suspendido (suspended): El proceso está temporalmente detenido, generalmente debido a la falta de recursos. No puede ejecutarse hasta que se reactive.

Transición	Descripción	Responsable
New → Ready	El proceso es creado y pasa al estado **Ready** cuando está preparado para ejecutarse.	**Long-Term Scheduler**
Ready → Running	El **Short-Term Scheduler** selecciona el proceso listo para asignarlo a la CPU.	**Short-Term Scheduler**
Running → Ready	El proceso es interrumpido (por ejemplo, al terminar su quantum o por un proceso de mayor prioridad).	**Short-Term Scheduler**
Running → Exit	El proceso ha completado su ejecución y pasa al estado **Exit (Terminated)**.	**Sistema Operativo**
Running → Blocked	El proceso está esperando un evento externo, como la finalización de una operación de E/S.	**Sistema Operativo**
Blocked → Ready	El evento esperado ha ocurrido (por ejemplo, una operación de I/O ha terminado).	**Sistema Operativo**
Blocked → Suspended	El proceso bloqueado es suspendido para liberar recursos del sistema.	**Medium-Term Scheduler**
Suspended → Ready	El proceso suspendido es reactivado y vuelve a estar listo para ejecutarse.	**Medium-Term Scheduler**



ix) ¿Qué scheduler de los mencionados en **1) f)** se encarga de las transiciones?

Short Term Scheduler:

- Responsabilidad: Selecciona qué proceso, de los que están en el estado listo (ready) será asignado a la CPU.
- Es llamado frecuentemente y debe ser rápido para minimizar la sobrecarga.

- De ready a running: Asigna un proceso listo a la CPU.
- De running a ready: Cuando el proceso en ejecución pierde la CPU por interrupciones o cambio de contexto.

Long Term Scheduler:

- Responsabilidad: Decide qué procesos nuevos (en el estado de new) se admiten al sistema y pasan al estado ready.
- Controla la carga del sistema ajustando la mezcla de procesos CPU Bound y I/O Bound.

- De new a ready: Admit procesos al sistema.

Medium Term Scheduler:

- Responsabilidad: Gestiona procesos que deben ser temporalmente suspendidos para optimizar el uso de recursos.
- Suele utilizarse en sistemas con memoria virtual para realizar swapping.

- De running o ready a suspended: Suspense procesos para liberar recursos.
- De suspended a ready: Reactiva procesos cuando los recursos están disponibles.

3) Para los siguientes algoritmos de scheduling:

- **FCFS** (First Come First Served)
- **SJF** (Shortest Job First)
- **Round Robin**
- **Prioridades**

a) Explique el funcionamiento mediante un ejemplo.
b) ¿Alguno de ellos requiere algún parámetro para su funcionamiento?
c) ¿Cuál es el más adecuado según los tipos de procesos y/o SO?
d) Cite ventajas y desventajas de su uso.

4) Para el algoritmo Round Robin, existen 2 variantes:

- Timer Fijo
- Timer Variable

a) ¿Qué significan estas 2 variantes?
b) Explique mediante un ejemplo sus diferencias.
c) En cada variante ¿Dónde debería residir la información del Quantum?

FCFS (First Come First Served)

- Los procesos son ejecutados en el orden que llegan (cola FIFO).
- No hay interrupciones: una vez que un proceso comienza, se ejecuta hasta su finalización.

Proceso	Llegada	CPU	~~Prioridad~~	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		TR	TE
P1	0	9	~~3~~	>1	2	3	4	5	6	7	8	9<																	9	0
P2	1	5	~~2~~		>								1	2	3	4	5<													13	8
P3	2	3	~~1~~			>												1	2	3<										15	12
P4	3	7	~~2~~				>														1	2	3	4	5	6	7<		21	14
FCFS			R Qeue	~~1~~	~~2~~	~~3~~	~~4~~																						14.5	8.5

- No requiere parámetros.

Ventajas:

- Sencillo de implementar.
- Justo para procesos que llegan primero.

Desventajas:

- Problema de convoy effect: procesos largos retrasan a los cortos.
- No es preemptive, no adecuado para sistemas interactivos.

SJF (Shortest Job First)

- Selecciona el proceso con el menor tiempo de ejecución primero.

- Puede ser preemptive o non preemptive.

Proceso	Llegada	CPU	~~Prioridad~~	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		TR	TE
P1	0	9	~~3~~	>1	2	3	4	5	6	7	8	9<																	9	0
P2	1	5	~~2~~		>											1	2	3	4	5<									16	11
P3	2	3	~~1~~			>							1	2	3<														10	7
P4	3	7	~~2~~				>														1	2	3	4	5	6	7<		21	14
SJF			Qeue	~~1~~	2	~~3~~	4																						14	8

- Como parámetro requiere el tiempo de ejecución de cada proceso.

Ventajas:

- Minimiza el tiempo promedio de espera.

Desventajas:

- Difícil de implementar si no se conoce la duración de los procesos.
- Problema de starvation para procesos largos.

<u>Round Robin Quantum fijo</u>

- Cada proceso se ejecuta durante un tiempo fijo llamado Quantum
- Si un proceso no finaliza dentro de su Quantum, regresa al final de la cola.

Proceso	Llegada	CPU	~~Prioridad~~	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		TR	TE
P1	0	9	~~3~~	>1	2	3	4									5	6	7	8					9<					**21**	**12**
P2	1	5	~~2~~		>			1	2	3	4									5<									16	11
P3	2	3	~~1~~			>						1	2	3<															9	6
P4	3	7	~~2~~				>								1						2	3	4		5	6	7<		21	14
RR-TF	**Q=4**		Qeue	~~1~~	~~2~~	~~3~~	4	~~1~~	~~2~~	4	~~1~~	4																	16.7	10.7

- Como parámetro requiere configurar un Quantum adecuado.

Ventajas:

- Simple de implementar.
- Justo para todos los procesos

Desventajas:

- El rendimiento depende del tamaño del Quantum.
  - Si es muy pequeño: demasiados cambios de contexto.
  - Si es muy grande: los procesos cortos son penalizados.

<u>Round Robin Quantum variable</u>

- El Quantum asignado a un proceso puede variar dependiendo de factores como:
  - Tamaño del proceso.
  - Prioridad.
  - Historial de ejecución

Proceso	Llegada	CPU	~~Prioridad~~	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		TR	TE
P1	0	9	~~3~~	>1	2	3	4												5	6	7	8					9<		**24**	**15**
P2	1	5	~~2~~		>			1	2	3	4												5<						19	14
P3	2	3	~~1~~			>						1	2	3<															9	6
P4	3	7	~~2~~				>								1	2	3	4						5	6	7<			20	13
RR-TV	**Q=4**		Qeue	~~1~~	~~2~~	~~3~~	4	~~1~~	~~2~~	4	~~1~~																		18	12

- Como parámetro requiere lógica para ajustar el Quantum dinámico.

Ventajas:

- Permite optimizar el rendimiento para diferentes tipos de procesos.

- Procesos largos pueden recibir más tiempo si tienen baja prioridad en el sistema.

Desventajas:

- Mayor complejidad en la implementación.
- Puede ser difícil ajustar los criterios de variación en el Quantum para satisfacer todos los procesos.

Quantum fijo

- El valor del Quantum es una constante predefinida en el sistema operativo.
- Está configurado como un parámetro global y afecta a todos los procesos por igual.

Quantum variable

- Puede ajustarse dinámicamente en función de:
  - **Prioridades**: Los procesos con mayor prioridad pueden recibir Quanta más largos.
  - **Historia de ejecución**: Procesos que han usado menos CPU pueden recibir Quanta más largos en iteraciones futuras.
  - **Requerimientos del proceso**: Basado en la naturaleza de cad proceso (I/O Bound o CPU Bound).
- En este caso, el Quantum es calculado por el sistema operativo en tiempo de ejecución y no es fijo.

El Quantum fijo o las reglas para el Quantum variable suelen residir en las estructuras del kernel, específicamente en:

- La cola de procesos listos.
- La tabla o estructura del scheduler.

En sistemas con Quantum variable, cada proceso puede tener su propio valor de Quantum asociado en su descriptor de proceso, generalmente almacenado en la estructura de datos que representa al proceso en el kernel (como el **PCB**).

Prioridades

- Cada proceso tiene una prioridad asociada.
- El proceso con la prioridad más alta se ejecuta primero.
- Puede ser preemptive o non preemptive.

Proceso	Llegada	CPU	Prioridad	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		TR	TE
P1	0	9	3	>1																2	3	4	5	6	7	8	9<		24	15
P2	1	5	2		>1				2	3	4	5<																	8	3
P3	2	3	1			>1	2	3<																					3	0
P4	3	7	2				>						1	2	3	4	5	6	7<										13	6
Prioridad			**Qeue1**	~~3~~																									12	6
			Qeue2	~~2~~	~~2~~	4																								
			Qeue3	~~1~~	~~1~~																									

- Como parámetro requiere definir prioridades para cada proceso.

Ventajas:

- Útil para sistemas donde ciertos procesos son más importantes que otros.

Desventajas:

- Problema de starvation: procesos de baja prioridad pueden nunca ejecutarse.

- Requiere un buen diseño para asignar prioridades.

Algoritmo	Uso más adecuado
FCFS (First Come First Served)	Ideal para sistemas simples por lotes donde el orden de llegada no afecta significativamente el rendimiento.
SJF (Shortest Job First)	Mejor para sistemas por lotes donde se pueden predecir los tiempos de ejecución de los procesos.
Round Robin con Quantum Fijo	Adecuado para sistemas interactivos con cargas de trabajo uniformes y donde los procesos tienen similar prioridad.
Round Robin con Quantum Variable	Óptimo para sistemas interactivos o mixtos donde los procesos tienen diferentes necesidades o prioridades.
Prioridades	Perfecto para sistemas donde ciertos procesos son más críticos (por ejemplo, sistemas en tiempo real).

9) Inanición (starving):

a) ¿Qué significa?

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

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

La inanición ocurre cuando un proceso no puede avanzar ni terminar su ejecución porque los recursos que necesita son constantemente asignados a otros procesos con mayor prioridad. Esto puede deberse a la política de scheduling o a la alta carga del sistema, dejando al proceso “hambriento” indefinidamente.

Algoritmos que pueden provocarla:

- **SJF** (Shortest Job First):
  - Si procesos cortos siguen llegando continuamente, los procesos más largos pueden quedar perpetuamente relajados.
- **Por prioridades**:
  - Procesos de baja prioridad pueden sufrir inanición si siempre hay procesos de mayor prioridad listos para ejecutarse.

Evitar la inanición:

- **SFJ**: Incrementar gradualmente la prioridad de un proceso cuanto más tiempo pase esperando en la cola. Esto asegura que los procesos más antiguos eventualmente sean ejecutados.
- **Por prioridades**:
    - **Envejecimiento** (aging): Similar al caso anterior, la prioridad de un proceso de baja prioridad aumenta con el tiempo para garantizar su ejecución.
    - **Quantum ajustado**: En sistemas apropiativos, los procesos de baja prioridad podrían recibir Quanta más largos para compensar su menor frecuencia de ejecución.