1g) El hecho de despachar representaria darle tiempo de CPU a un proceso para que se ejecute. Se despacha un proceso que se encuentra en estado de ready.

2) a)
Aquí tienes una descripción detallada de cada uno de los comandos que mencionaste en Linux Debian:

i. `top`:
- Descripción: `top` es una herramienta de línea de comandos que proporciona una visión en tiempo real de los procesos en ejecución en el sistema. Muestra información sobre el uso de la CPU, la memoria y otros recursos del sistema. Es útil para monitorear el rendimiento del sistema y la utilización de recursos.

ii. `htop`:
- Descripción: `htop` es una versión mejorada de `top`. Ofrece una interfaz más amigable y funcionalidad adicional. Permite navegar fácilmente por la lista de procesos, proporciona una vista de árbol de los procesos y permite realizar acciones como matar procesos directamente desde la interfaz.

iii. `ps`:
- Descripción: `ps` es un comando que muestra una lista de los procesos en ejecución en el sistema. Puede mostrar información detallada sobre los procesos, como su identificador de proceso (PID), el usuario que los ejecuta, el uso de recursos y otros detalles.

iv. `pstree`:
- Descripción: `pstree` muestra una representación jerárquica de los procesos en el sistema en forma de un árbol. Esto ayuda a comprender las relaciones entre los procesos y sus padres o hijos.

v. `kill`:
- Descripción: `kill` es un comando utilizado para enviar señales a procesos en ejecución. Por defecto, envía la señal SIGTERM, que solicita amablemente a un proceso que se cierre. También se puede utilizar para enviar señales más fuertes para forzar la terminación de un proceso.

vi. `pgrep`, `pkill`, `killall`:
- `pgrep` busca procesos basados en patrones de nombres o comandos y muestra sus PIDs.
- `pkill` envía señales a procesos basados en patrones de nombres o comandos.
- `killall` envía señales a procesos basados en nombres de procesos. Nota que este comando no es el mismo que `killall` en algunos otros sistemas, que se usa para matar todos los procesos de un usuario. En Debian, `killall` se utiliza para enviar señales a procesos individuales.

vii. `killall`:
- Descripción: `killall` en Debian no es el mismo que en algunas otras distribuciones. En Debian, `killall` se usa para enviar señales a procesos individuales basados en nombres de proceso, en lugar de matar todos los procesos de un usuario.

viii. `renice`:
  - Descripción: `renice` es un comando que permite cambiar la prioridad de un proceso en términos de la asignación de CPU. Puedes aumentar o disminuir la prioridad de un proceso para controlar su uso de recursos de CPU.

ix. `xkill`:
  - Descripción: `xkill` es un comando que se utiliza en entornos gráficos de X Window System. Permite seleccionar una ventana en la pantalla y cerrarla forzosamente, lo que puede ser útil cuando una aplicación deja de responder.

x. `atop`:
  - Descripción: `atop` es una herramienta de monitoreo de recursos avanzada que proporciona información detallada sobre el uso de CPU, memoria, disco y otros recursos. Ofrece una interfaz interactiva similar a `top` pero con más características y una mejor capacidad de registro.

La disponibilidad de estos comandos puede variar según la distribución de Debian y si están instalados en el sistema. Puedes instalar algunos de estos comandos adicionales a través del gestor de paquetes de Debian, como `apt` o `apt-get`, si no están disponibles de forma predeterminada.

b)

```c
#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;
}
```

Este código es un programa simple en C que utiliza llamadas al sistema para crear procesos hijos. Aquí está el flujo de ejecución y la explicación paso a paso:

1. Se incluyen tres archivos de encabezado:
  - `<stdio.h>` para las funciones de entrada y salida estándar.
  - `<sys/types.h>` para definiciones de tipos, incluyendo `pid_t`.

- `<unistd.h>` para llamadas al sistema, como `fork()`.

2. Se define la función `main`, que no recibe argumentos (`void`) y devuelve un entero (`int`).

3. Se declara una variable entera `c`.

4. Se declara una variable `pid` de tipo `pid_t`. Esta variable se utilizará para almacenar el valor de retorno de la función `fork()`.

5. Se imprime "Comienzo." en la salida estándar usando `printf()` para indicar el inicio del programa.

6. Se inicia un bucle `for` que se ejecutará tres veces (c < 3).

7. Dentro del bucle, se llama a la función `fork()`. La función `fork()` crea un nuevo proceso hijo. Cada vez que se llama a `fork()`, se crea un nuevo proceso hijo y ambos (padre e hijo) continúan ejecutándose desde el punto en que se llamó a `fork()`. En este caso, se crean tres procesos hijos en total (cada uno de los cuales también crea tres procesos, debido a la repetición del bucle).

8. Después de completar el bucle `for`, el programa llega a la línea `printf("Proceso\n")`. Sin embargo, esta línea se ejecuta una sola vez, independientemente de cuántos procesos hijos se hayan creado. La razón es que esta línea está fuera del bucle `for`.

9. Finalmente, el programa devuelve 0 (`return 0;`) y termina.

En resumen, este programa imprime "Comienzo." una vez al inicio, crea nueve procesos hijos en el bucle `for` (3 iteraciones, cada una creando 3 procesos), pero luego imprime "Proceso" solo una vez al final. Cada proceso hijo creado por `fork()` hereda el flujo de ejecución del proceso padre, lo que puede generar múltiples procesos, pero la línea de impresión "Proceso" solo se ejecuta una vez en el proceso padre.

Este código es un programa en C que utiliza la función `fork()` para crear múltiples procesos. Aquí está el análisis de las preguntas:

i. ¿Cuántas líneas con la palabra "Proceso" aparecen al final de la ejecución de este programa?.

Al final de la ejecución del programa, solo aparece una línea con la palabra "Proceso". Esto se debe a que la línea `printf("Proceso\n");` se encuentra fuera del bucle `for` y, por lo tanto, se ejecuta solo una vez.

ii. ¿El número de líneas es el número de procesos que han estado en ejecución?.

No, el número de líneas impresas con la palabra "Proceso" no es igual al número de procesos creados en el bucle `for`. En este programa, se crean tres procesos en el bucle `for`, pero la línea `printf("Proceso\n");` se ejecuta solo una vez después de que el bucle

haya terminado. Por lo tanto, solo se imprime una línea con la palabra "Proceso" al final de la ejecución, independientemente del número de procesos creados.

c)

d)
La comunicación entre procesos a través de pipes es una forma de intercambio de datos entre procesos en un sistema operativo. Los pipes son una de las formas más simples de comunicación entre procesos y se utilizan para transmitir datos desde un proceso emisor (escritor) a un proceso receptor (lector). Aquí tienes respuestas a tus preguntas específicas:

i. Forma de comunicación entre procesos a través de pipes:
- Los pipes son una técnica de comunicación entre procesos que permite la transferencia de datos de un proceso a otro.
- Se utilizan principalmente para comunicación entre un proceso padre y sus procesos hijos o entre dos procesos relacionados.
- Un pipe se comporta como un canal unidireccional, lo que significa que los datos fluyen en una dirección específica, generalmente desde el proceso escritor al proceso lector.
- Los pipes son útiles para implementar tuberías en la línea de comandos y para compartir datos entre procesos relacionados.

ii. Creación de un pipe en C:
- En C, los pipes se crean utilizando la función `pipe()`, que está disponible en la biblioteca `<unistd.h>`.
- La función `pipe()` crea un par de descriptores de archivo (file descriptors) que se pueden utilizar para la comunicación entre procesos.
- Un descriptor de archivo se utiliza para escribir datos en el pipe (escritor), y el otro se utiliza para leer datos desde el pipe (lector).
- Aquí hay un ejemplo de cómo se crea un pipe en C:

```c
#include <unistd.h>

int mypipe[2];
if (pipe(mypipe) == -1) {
    // Manejo de error si la creación del pipe falla
}
```

iii. Parámetros necesarios para la creación de un pipe y su uso:
- La función `pipe()` no toma ningún parámetro en sí. Simplemente llama a la función para crear el pipe y devuelve un par de descriptores de archivo en un arreglo.
- El arreglo `mypipe` en el ejemplo anterior contiene dos descriptores de archivo: `mypipe[0]` para lectura y `mypipe[1]` para escritura.

- Los descriptores de archivo se utilizan para enviar y recibir datos a través del pipe. Un proceso escribe datos en el descriptor de escritura y otro proceso lee esos datos del descriptor de lectura.
- El descriptor de lectura y el descriptor de escritura son necesarios para que ambos procesos se comuniquen a través del pipe.
- Es importante cerrar los descriptores de archivo no utilizados adecuadamente en cada proceso para evitar posibles problemas de bloqueo y fuga de recursos.

iv) Los pipes en sistemas Unix y Linux permiten la comunicación unidireccional entre procesos. Específicamente, los pipes admiten la comunicación entre un proceso escritor (productor) y un proceso lector (consumidor)

e) El sistema operativo necesita conocer cierta información esencial sobre un proceso para gestionarlo adecuadamente. La información mínima que el sistema operativo debe tener sobre un proceso incluye:

1. **Identificación del proceso (PID)**: Cada proceso se identifica de manera única mediante un número llamado PID (Process ID). El PID permite al sistema operativo y a otros procesos referirse al proceso de manera unívoca.

2. **Estado del proceso**: El sistema operativo debe mantener información sobre el estado del proceso, es decir, si el proceso está en ejecución, listo para ejecutar, bloqueado o terminado.

3. **Contexto del proceso**: Esto incluye la información necesaria para reanudar la ejecución del proceso en el punto donde se detuvo, como los registros de la CPU, el contador de programa, el contenido de la memoria y otros registros de estado. Este contexto se almacena en una estructura de datos llamada PCB (Process Control Block), que se asocia a cada proceso.

4. **Recursos asignados**: El sistema operativo debe llevar un registro de los recursos asignados a un proceso, como descriptores de archivo, memoria, identificadores de archivos abiertos, identificadores de sockets, etc.

5. **Relaciones de padre-hijo**: El sistema operativo debe registrar las relaciones de parentesco entre procesos. Un proceso padre crea procesos hijos, y esta información es útil para la gestión y terminación de procesos.

La estructura de datos asociada a la cual se almacena esta información es el "Process Control Block" (PCB), que es una estructura de datos que contiene todos los datos necesarios para gestionar un proceso. El PCB almacena información sobre el proceso, como el PID, el estado, el contexto de la CPU, los recursos asignados y otros detalles relevantes. Cada proceso tiene su propio PCB, y el sistema operativo mantiene una lista de PCBs para gestionar todos los procesos en el sistema. La información almacenada en el PCB permite al sistema operativo cambiar de un proceso a otro, gestionar la concurrencia y garantizar que los recursos se utilicen adecuadamente.

f)

CPU bound: son aquellos donde solamente deben ejecutar rafagas de cpu, es decir que en todo su tiempo de ejecución van a usar tiempo de la cpu.
I/O bound: Son aquellos que van combinando rafagas de CPU con rafagas de E/S. Es decir que dsp de ejecutarse un tiempo en la cpu van a tener q hacer una operacion de e/s por lo que van a tener q abandonar la cpu.

g) Los estados de la imagen, new …. etc.

h) **Los procesos tienen distintos estados cuando se van ejecutando**, lo que lo hace mas sencillo de gestionar y administrar.
Cuando un proceso padre crea un proceso hijo entra en un estado de new. Luego se verifica si puede ser admitido en memoria (en la ready), luego de esto va a quedar esperando y compitiendo en memoria ram por la cpu hasta que un planificador (short term scheduler) lo seleccione como candidato a ser despachado hacia el estado de running (asignarle tiempo de cpu para que el proceso se pueda ejecutar). Un proceso podría terminar su ejecución en una ráfaga de ejecución sola o puede volver al estado de Ready y seguir compitiendo por la cpu desde la memoria principal, por ejemplo por alguna interrupción o que se le acabó el tiempo de ejecución. Otro caso es que ocurra un evento de espera (vinculado a E/S) entonces va a un estado de bloqueo (puede estar en la memoria ram o en la swap) en el cual puede ocurrir un evento que genere que pase a estado de ready a competir nuevamente, o puede pasar a estado de suspendido (en el cual seria fuera de la swap) y tendria que ser reactivado por otro planificador para volver al estado ready.

i)
Las transiciones de un proceso entre diferentes estados (como pasar de listo a en ejecución o de en ejecución a bloqueado) son gestionadas principalmente por el "scheduler de corto plazo" (short-term scheduler), que también se conoce comúnmente como el "planificador de CPU" o "planificador de ejecución". Este planificador se encarga de tomar decisiones muy frecuentes y rápidas sobre qué proceso debe ejecutarse en la CPU en un momento dado, basándose en la asignación de tiempo de CPU y las prioridades, con el objetivo de maximizar la utilización de la CPU y garantizar una ejecución justa y eficiente de los procesos.

3)
Voy a explicar brevemente el funcionamiento de cada uno de los algoritmos de scheduling que mencionaste, así como responder a tus preguntas:

FCFS (First-Come-First-Served):

(a) Funcionamiento: FCFS asigna la CPU al primer proceso que llega a la cola de listos y se encuentra esperando. Es un algoritmo no preemptivo, lo que significa que un proceso ejecuta hasta su finalización antes de que otro proceso pueda tomar el control.

Ejemplo: Supongamos que tenemos tres procesos A, B y C, en ese orden, llegando a la cola de listos. FCFS ejecutará A primero, luego B y finalmente C.

(b) Parámetros: No requiere parámetros adicionales para su funcionamiento. Simplemente sigue el orden de llegada.

(c) Adecuación: FCFS es adecuado para procesos de igual prioridad y donde la equidad en la asignación de CPU es importante. Sin embargo, no es adecuado para sistemas en los que algunos procesos pueden ser largos, ya que los procesos largos pueden bloquear procesos más cortos y causar ineficiencias.

(d) Ventajas y desventajas:
- Ventajas: Simple y fácil de entender. Evita la inanición.
- Desventajas: Puede llevar a tiempos de espera largos y baja utilización de la CPU.

SJF (Shortest Job First):

(a) Funcionamiento: SJF asigna la CPU al proceso más corto en tiempo de ejecución. Esto minimiza el tiempo de espera promedio y maximiza la eficiencia.

Ejemplo: Si tenemos tres procesos A (3 ms), B (4 ms) y C (2 ms), SJF ejecutará primero C, luego A y finalmente B.

(b) Parámetros: Requiere conocer el tiempo de ejecución de cada proceso, lo que a menudo no es predecible en sistemas en tiempo real.

(c) Adecuación: SJF es adecuado para entornos donde es posible estimar o predecir con precisión los tiempos de ejecución. Puede no ser adecuado para sistemas en tiempo real donde los tiempos de ejecución son variables y cambiantes.

(d) Ventajas y desventajas:
- Ventajas: Minimiza el tiempo de espera promedio y mejora la eficiencia.
- Desventajas: Requiere información precisa sobre los tiempos de ejecución, lo que a menudo es difícil de obtener.

Round Robin:

(a) Funcionamiento: Round Robin asigna la CPU a cada proceso en la cola de listos durante un período de tiempo fijo llamado "quantum". Cuando se agota el quantum, el proceso se mueve al final de la cola y se da tiempo a otro proceso. Este es un algoritmo preemptivo.

Ejemplo: Con un quantum de 10 ms y tres procesos A, B y C, Round Robin asignará 10 ms a A, luego 10 ms a B y finalmente 10 ms a C. Luego, el ciclo se repite.

(b) Parámetros: Requiere definir el tamaño del quantum, que puede afectar el rendimiento.

(c) Adecuación: Round Robin es adecuado para sistemas en los que se necesita equidad y previsibilidad en la asignación de CPU, pero puede no ser adecuado para procesos con requisitos de tiempo real estrictos debido a la fragmentación temporal.

(d) Ventajas y desventajas:
- Ventajas: Proporciona equidad en la asignación de CPU y es fácil de implementar.

- Desventajas: Puede haber una sobrecarga de cambio de contexto y puede no ser eficiente para procesos largos.

Prioridades:

(a) Funcionamiento: El algoritmo de prioridades asigna la CPU al proceso con la prioridad más alta. Los procesos se ejecutan en orden de prioridad.

Ejemplo: Si tenemos tres procesos A (prioridad alta), B (prioridad media) y C (prioridad baja), el algoritmo de prioridades ejecutará primero A, luego B y finalmente C.

(b) Parámetros: Requiere asignar prioridades a cada proceso, lo que puede ser complicado y requerir ajustes.

(c) Adecuación: Las prioridades son adecuadas para sistemas donde ciertos procesos deben tener prioridad sobre otros, como en sistemas en tiempo real. Sin embargo, se deben manejar cuidadosamente para evitar inversiones de prioridad y otros problemas.

(d) Ventajas y desventajas:
- Ventajas: Permite asignar prioridad a procesos críticos. Puede ser útil en sistemas en tiempo real.
- Desventajas: Puede llevar a inversiones de prioridad y requerir una gestión precisa de las prioridades.

La elección del algoritmo de scheduling depende de las características del sistema y los tipos de procesos que se ejecutan en él. Ningún algoritmo es óptimo para todas las situaciones, y la elección se basa en los requisitos específicos y las limitaciones del sistema.

4) excel.

5) excel.
d) Si vos tenes un quantum muy pequeño, la cpu va a estar mucho tiempo haciendo cambio de contexto (guardando el contexto de un proceso que se saca de la cpu y cargando el contexto de un proceso nuevo). Y esto genera overhead, que significa que estás mas tiempo haciendo este proceso de expulsar y cargar procesos que estar realmente ejecutando los procesos.
Si el quantum es muy grande, un valor de quantum mayor a los ciclos de cpu que necesitan mis procesos lo que generas es el primer algoritmo FCFS (te va a quedar como si hubieses aplicado ese algoritmo) ya que jamas se va a acabar el quantum antes de que el proceso termine, no estas haciendo nada con el quantum.

e)
La elección del valor del quantum en el algoritmo Round Robin (RR) es importante y debe basarse en las características y requisitos específicos del sistema. Aquí hay algunas consideraciones sobre cuándo utilizar un valor de quantum alto y las ventajas y desventajas asociadas:

Utilizar un valor de quantum alto (ventajas y desventajas):

Ventajas de un valor de quantum alto:

1. **Menor sobrecarga de cambio de contexto:** Un valor de quantum más grande significa que los procesos se ejecutan durante más tiempo antes de cambiar a otro proceso. Esto reduce la cantidad de cambios de contexto, lo que es beneficioso en términos de rendimiento, ya que los cambios de contexto pueden ser costosos en términos de tiempo de CPU y recursos del sistema.

2. **Menos fragmentación temporal:** Un valor de quantum más grande reduce la fragmentación temporal, que es el espacio desperdiciado entre los procesos debido a cambios frecuentes. Esto mejora la eficiencia en la asignación de CPU.

3. **Menos sobrecarga de planificación:** Con un valor de quantum más grande, la sobrecarga de planificación (overhead) relacionada con la planificación y el cambio de contexto es menor en comparación con valores más pequeños.

Desventajas de un valor de quantum alto:

1. **Tiempo de respuesta más lento:** Un valor de quantum alto puede llevar a tiempos de respuesta más lentos para procesos interactivos o procesos cortos que necesitan una rápida asignación de CPU.

2. **Posible ineficiencia en procesos largos:** Los procesos largos pueden bloquear la CPU durante un tiempo prolongado antes de dar la oportunidad a otros procesos de ejecutarse, lo que puede causar ineficiencia.

3. **Menos equidad en la asignación de CPU:** A medida que aumenta el valor de quantum, la equidad en la asignación de CPU disminuye. Los procesos largos tienen más tiempo para ejecutar, lo que puede llevar a la ineficiencia y a que los procesos más cortos esperen más tiempo en la cola de listos.

En resumen, un valor de quantum alto en el algoritmo Round Robin es beneficioso en términos de reducir la sobrecarga de cambio de contexto y fragmentación temporal, pero puede resultar en tiempos de respuesta más lentos para procesos interactivos y menos equidad en la asignación de CPU. La elección del valor de quantum debe ser una decisión equilibrada que considere las necesidades del sistema y los tipos de procesos que se ejecutan en él.

7)
b) Consultar. Con respecto al algoritmo SJF, el algoritmo SRTF tiene mucho mejor promedio, sobre todo con respecto al tiempo de espera.

8) Excel.
c) El algoritmo de planificación basado en prioridades es un enfoque en el que se asigna la CPU a los procesos en función de su prioridad relativa. Cada proceso se asocia con una prioridad, y el proceso con la prioridad más alta recibe la CPU. Aquí están las ventajas y

desventajas de este enfoque, junto con las circunstancias en las que sería apropiado y las situaciones en las que podría no ser relevante:

Ventajas del algoritmo de prioridades:

1. **Adaptabilidad:** Permite asignar recursos de CPU de acuerdo con la importancia y la urgencia de los procesos. Esto es valioso en entornos donde algunos procesos son más críticos que otros.

2. **Adecuado para sistemas en tiempo real:** Los sistemas en tiempo real a menudo utilizan algoritmos de prioridades para garantizar que los procesos más críticos cumplan con los plazos. Es crucial en aplicaciones médicas, sistemas de control de tráfico aéreo, etc.

3. **Personalización:** Los administradores del sistema pueden ajustar las prioridades de los procesos según las necesidades del sistema.

4. **Equidad:** Puede proporcionar equidad en la asignación de CPU, garantizando que los procesos con prioridad más alta reciban recursos adecuados.

Desventajas del algoritmo de prioridades:

1. **Riesgo de inanición:** Si se asignan prioridades altas de manera continua a ciertos procesos, los procesos con prioridades más bajas pueden quedarse esperando indefinidamente (inanición).

2. **Posible abuso:** Si los usuarios pueden especificar sus propias prioridades, puede haber un abuso de recursos y un impacto en la equidad.

3. **Sobrecarga de gestión de prioridades:** La gestión de prioridades puede ser compleja y requerir un seguimiento y ajuste constantes, lo que puede aumentar la sobrecarga del sistema.

Circunstancias en las que utilizaría el algoritmo de prioridades:

- En sistemas en tiempo real o aplicaciones críticas donde es necesario cumplir con plazos estrictos.
- Cuando es esencial dar prioridad a procesos críticos, como tareas de control, monitoreo de seguridad o tareas médicas.
- En sistemas que requieren un equilibrio entre la equidad y la importancia de los procesos, y donde los administradores pueden gestionar las prioridades de manera efectiva.

Situaciones en las que la implementación de prioridades podría no ser relevante:

- En sistemas donde todos los procesos son igualmente importantes y no se requiere una priorización específica.
- Cuando los procesos no tienen un grado claro de importancia o cuando la gestión de prioridades agregaría una sobrecarga innecesaria al sistema.

- En sistemas donde los plazos no son críticos y la eficiencia en la utilización de la CPU es más importante que la priorización de procesos.

9) a) La "inanición" (starvation en inglés) es un problema en la planificación de procesos que ocurre cuando un proceso o un conjunto de procesos no obtienen la oportunidad de ejecutarse debido a la priorización constante de otros procesos. En otras palabras, los procesos en estado de espera en una cola de listos pueden quedar atrapados indefinidamente sin recibir la CPU, lo que puede causar un retraso significativo o incluso un bloqueo total de esos procesos. La inanición es un problema no deseado en la planificación de procesos, ya que puede afectar negativamente el rendimiento y la equidad del sistema.

b) Lo pueden provocar, SJF, SRTF, PRORIDADES.(los 2 primeros para procesos largos, y el ultimo para procesos con baja prioridad).

c) Para mitigar la inanición en el algoritmo SJF, algunas implementaciones pueden incluir mecanismos de envejecimiento. Estos mecanismos aumentan gradualmente la prioridad de los procesos en espera, lo que garantiza que, con el tiempo, todos los procesos tengan la oportunidad de ejecutarse, incluso si hay procesos largos en la cola de listos.

Para el algoritmo de prioridades, se puede usar lo que se llama Aging o Penalty, la cual permite a un proceso cambiar su prioridad durante su ciclo de vida (aumentarla o decrementarla).

La inanición en el algoritmo Round Robin (RR) generalmente ocurre con la asignación de un tiempo fijo (quantum) a cada proceso. Con un quantum fijo, los procesos largos pueden ocupar la CPU durante un tiempo prolongado y, como resultado, los procesos más cortos pueden quedar en espera indefinidamente, lo que se considera una forma de inanición. Para solucionar o mitigar la inanición en el algoritmo RR con un tiempo fijo, se pueden implementar varias estrategias:

Ajustar el tamaño del quantum: Utilizar un quantum más pequeño puede reducir la inanición, ya que permite a los procesos más cortos obtener acceso más rápido a la CPU. Sin embargo, un quantum muy pequeño puede aumentar la sobrecarga de cambio de contexto, por lo que se debe encontrar un equilibrio.

Implementar envejecimiento: La técnica de envejecimiento implica aumentar el quantum o la prioridad de los procesos en espera con el tiempo. Esto asegura que los procesos largos eventualmente cedan la CPU a los procesos más cortos. El envejecimiento puede ser efectivo para mitigar la inanición.

Utilizar Round Robin con quantum variable: En lugar de un quantum fijo, se puede implementar un quantum variable que se ajusta en función de la duración de ejecución de los procesos. Esto ayuda a garantizar que los procesos más cortos tengan la oportunidad de ejecutarse antes que los procesos más largos.

Prioridad dinámica: Algunas implementaciones de RR utilizan prioridades dinámicas, donde los procesos con más tiempo de ejecución acumulado o aquellos que han esperado más tiempo en la cola de listos obtienen un quantum más corto. Esto equilibra la asignación de CPU y reduce la inanición.

Utilizar colas de prioridad: En lugar de una cola de listos única, se pueden utilizar colas de prioridad múltiples para gestionar procesos con diferentes niveles de

prioridad. Esto permite que los procesos de prioridad más alta se ejecuten antes, lo que reduce el riesgo de inanición.

10) El “R queue que seria? la cola de listos?.. COnsultar logica del inciso B, si esta bien el orden en el que cargo los procesos a las colas, y si está bien la logica en el cual no puede haber en un mismo ciclo de CPU más de 1 operación de E/S de un recurso en particular (tipo tener 2 R1). Ademas, en la cola de listos (la primera) en vez de poner 1 y luego 3, puse al revés, pero dsp creo q vi porque era, preguntar si es 1 y luego 3 o 3 y luego 1. Despues consultar la ultima tabla tmb.

11)

(a) Round Robin

Beneficia a los procesos ligados a CPU. Los procesos E/S podrían, en un sólo quantum utilizar sólo una parte del mismo antes de bloquearse para esperar un proceso de E/S, y no volverán a la cola de listos hasta que dicha operación termine. Los procesos CPU bound, por otro lado, sólo abandonan el CPU por haberse acabado su quantum (o el propio proceso), y ni bien esto sucede retornan a la cola de listos, requiriendo esperar menos tiempo por el uso del procesador.

Un proceso I/O bound usara poco tiempo de CPU antes de esperar a que se complete la E/S y round robin lo pondrá al final de la cola incluso si usó poco tiempo. Un proceso CPU Bound va a usar la CPU aprovechando todo el Quantum.

(b) SRTF (Shortest Remaining Time First)

Favorece a los procesos ligados a E/S, ya que estos por principio hacen menos uso de CPU, con lo que desde el momento en que llegan a la cola de listos tienen más posibilidades de acceder al procesador. Por otro lado, al requerir ráfagas cortas de CPU, cuentan con más chances de hacer uso de la totalidad de su ráfaga y esperar al nuevo proceso de E/S (o terminarse) antes de ser expulsados.

12) CONSULTARLO, VER ANOTACION Q MARQUE AHI Y COMPLETARLO.

13)
En un algoritmo de Round Robin Variable (VRR) con Timer Variable, el quantum de un proceso puede ser variable y se ajusta según el tiempo transcurrido desde que se ejecutó por última vez. Este enfoque tiene como objetivo mejorar la eficiencia al asignar más tiempo de CPU a procesos que han estado esperando en la cola durante más tiempo. A pesar de que el quantum se ajusta dinámicamente, es posible que un proceso nunca llegue a un quantum de 0 en algunos casos. Aquí hay una explicación de por qué esto podría suceder:

1. Prioridades o envejecimiento de procesos: Si el SO utiliza prioridades o un mecanismo de envejecimiento de procesos para determinar cuál proceso se ejecutará a continuación, un proceso con una prioridad más alta o uno que ha estado esperando en la cola durante mucho tiempo podría recibir un quantum más largo en comparación con otros procesos. Esto podría hacer que su quantum sea suficientemente grande como para que no llegue a 0 antes de que se seleccione nuevamente.

2. Interrupciones del temporizador: El quantum de un proceso se decrementa cada vez que ocurre una interrupción de reloj. Si las interrupciones de reloj ocurren a una velocidad baja o si el temporizador se establece inicialmente con un valor grande, un proceso podría no agotar su quantum antes de que ocurra la próxima interrupción de reloj. Esto podría hacer que el quantum de ese proceso nunca llegue a 0.

3. Procesos bloqueados: Si un proceso se bloquea durante su ejecución, su quantum puede detenerse y restaurarse cuando se desbloquea. Esto puede hacer que el quantum del proceso nunca llegue a 0, ya que la ejecución se detiene temporalmente mientras está bloqueado.

En resumen, bajo un algoritmo de VRR con Timer Variable, es posible que el quantum de un proceso nunca llegue a 0 debido a la dinámica de asignación de tiempos y las interrupciones del temporizador. Esto no es necesariamente un problema, ya que el objetivo del algoritmo es proporcionar un equilibrio entre la equidad en la asignación de recursos y la eficiencia en la ejecución de procesos.

14) git.

15)a) Para los procesos interactivos > los RR.
Para los procesos batch el FCFS.
B) Usaria el algoritmo de prioridades con AGING.

16) pedir ayuda a profe para hacerlo pq es un quilombo.

17) se hace en base al 16.

18)
Para implementar un algoritmo de planificación que tome en cuenta el tiempo de ejecución consumido por los procesos y penalice a aquellos que han estado en ejecución durante más tiempo, puede utilizar un esquema de Colas Multinivel con Realimentación (Multilevel Feedback Queue Scheduling). Este enfoque permite asignar prioridades dinámicamente en función del tiempo de ejecución de los procesos y evitar la inanición de procesos con una planificación justa. Aquí está cómo podría configurar el sistema:

1. **Colas Multinivel**: Utilice múltiples colas para organizar los procesos en función de su tiempo de ejecución. Puede configurar, por ejemplo, tres colas:

   a. **Cola de Alta Prioridad (Queue 1)**: Procesos que han consumido poco tiempo de CPU.
   b. **Cola de Prioridad Media (Queue 2)**: Procesos que han consumido una cantidad moderada de tiempo de CPU.
   c. **Cola de Baja Prioridad (Queue 3)**: Procesos que han consumido una cantidad significativa de tiempo de CPU.

2. **Algoritmo para Cada Cola**:

   a. **Cola de Alta Prioridad (Queue 1)**: Utilice un algoritmo de planificación SJF (Shortest Job First) para esta cola. Los procesos con menor tiempo de ejecución pendiente se ejecutarán primero. Esto penaliza a los procesos más largos en ejecución y evita que ocupen esta cola por mucho tiempo.

   b. **Cola de Prioridad Media (Queue 2)**: Utilice un algoritmo Round Robin para esta cola. Cada proceso obtendrá un quantum de tiempo para ejecutarse, lo que garantiza una equidad razonable y permite a los procesos más largos de ejecución avanzar, aunque con un quantum limitado.

   c. **Cola de Baja Prioridad (Queue 3)**: En esta cola, puede usar un algoritmo Round Robin con un quantum más largo o incluso un algoritmo FCFS (First-Come, First-Served). Los procesos que han consumido más tiempo de CPU se asignan a esta cola y se ejecutan cuando no hay procesos más cortos en las colas anteriores.

3. **Administración de las Colas entre Sí**: Para evitar la inanición, los procesos deben moverse entre las colas con el tiempo. Aquí hay algunas reglas para administrar la promoción y la degradación de procesos entre colas:

   - Cuando un proceso en Queue 1 ha agotado su quantum, puede moverse a Queue 2.
   - Cuando un proceso en Queue 2 ha agotado su quantum, puede moverse a Queue 3.
   - Si un proceso en Queue 2 o Queue 3 bloquea o se libera, puede regresar a Queue 1 si tiene un tiempo de ejecución pendiente suficientemente bajo.
   - Los procesos que permanecen en Queue 3 durante un tiempo prolongado pueden eventualmente regresar a Queue 2 o 1 si se desea.

Estas reglas permiten que los procesos con menos tiempo de CPU pendiente se ejecuten con prioridad y evitan la inanición al permitir que los procesos en colas de prioridad más baja tengan la oportunidad de ejecutarse. Este enfoque proporciona una planificación justa mientras penaliza a los procesos más largos de ejecución.

GIT:

En general, a un proceso se le concede un tiempo T de permanencia en una cola, cuando lo supera, pasará a la cola inmediatamente inferior con menor prioridad, es decir, se disminuirá su prioridad en una unidad. La elección del valor que se le dará al tiempo T varía mucho de un sistema a otro, depende del número de procesos existentes, del tipo de procesos y del número de colas.

Se pueden usar mecanismos de envejecimiento para evitar el bloqueo indefinido de un proceso, estos mecanismos consisten en incrementar la prioridad de los procesos que estén demasiado tiempo esperando en una cola de prioridad baja, para pasarlos a una cola de prioridad más alta y que se puedan ejecutar antes.

En resumen, este algoritmo se puede definir por los siguientes parámetros:

- El número de colas.
- El algoritmo de planificación de cada cola.
- El algoritmo de planificación entre las distintas colas.
- El método usado para determinar cuándo pasar un proceso a una cola de prioridad más alta.
- El método usado para determinar cuándo pasar un proceso a una cola de prioridad más baja.
- El método usado para determinar en qué cola se introducirá un proceso cuando haya que darle servicio.

Es como que tenes distintas colas de prioridades, donde la de mas arriba ejecutan procesos que tienen menos tiempo de cpu consumidos, en la cual estos se ejecutan primero, luego los de las colas de mas abajo que tienen menos prioridades, y que tiene la cuestión de la realimentación que permite incrementar y decrementar la prioridad de los procesos para evitar la inanición básicamente.

19 y 20) git.

21)

21. Explicar porqué si el quantum "q.en Round-Robin se incrementa sin límite, el método se aproxima a FIFO.

Porque al ser el Quantum tan largo, va a llegar un punto donde los procesos saldrán por su propia cuenta (finalizaron o por E/S), antes de ser expulsado. De esta forma se eliminaría por completo la rotación de procesos, ya que cada proceso encolado acabaría por sí mismo sin regresar a la cola de ejecución a esperar su siguiente turno (porque no lo necesitaría, salvo que se abandone el procesador por E/S).

Basicamente, el quantum seria mayor que el tiempo de cpu que necesitaria cualquier proceso para ejecutarse, por lo que nunca llegaria a expulsar a un proceso, sino que este se ejecutará hasta que termine o sea interrumpido por una E/S, por ende la funcionalidad de tener un quantum no la estarias usando para nada y te quedaria un caso igual al de FIFO.

22)

(a) Las PCs de escritorio habituales se asocian con sistemas homogéneos y multiprocesadores débilmente acoplados. En una PC típica, los procesadores (por ejemplo, CPU) suelen ser iguales en capacidad y arquitectura, lo que los clasifica como homogéneos. Además, en una PC, cada procesador tiene su propia memoria principal y no comparten directamente recursos de memoria ni canales. Cada núcleo de la CPU en una PC de escritorio tiene acceso independiente a su propia memoria RAM.

(b) La asignación de procesos de manera simétrica significa que todos los procesadores de un sistema multiprocesador homogéneo tienen igualdad de condiciones en términos de recursos y capacidad de procesamiento. No hay un procesador principal o dominante, y la asignación de tareas se realiza de manera equitativa entre todos los procesadores. Cada procesador tiene la misma capacidad para ejecutar cualquier tarea y no hay jerarquía en términos de poder de procesamiento.

(c) Trabajar bajo un esquema maestro/esclavo implica que en un sistema multiprocesador, uno o más procesadores (maestros) tienen un papel de control y coordinación sobre otros procesadores (esclavos). El maestro se encarga de la administración de tareas, la asignación de recursos y la coordinación de las operaciones de los esclavos. Los esclavos realizan el trabajo asignado por el maestro y pueden ser procesadores especializados en tareas específicas, como procesadores de E/S o unidades de cómputo dedicadas. Este enfoque es común en sistemas en los que se necesita una administración centralizada y coordinación de tareas, como en clústeres de servidores o sistemas distribuidos.

23) git.

24)a)
ˆ Huella: estado que el proceso va dejando en la cache de un procesador
ˆ Afinidad: preferencia de un proceso para ejecutar en un procesador. Afinidad con un procesador: La afinidad con un procesador se refiere a la preferencia o restricción de un proceso para ejecutarse en un procesador específico. Esto significa que un proceso puede estar configurado para ejecutarse en un núcleo o procesador particular en un sistema multiprocesador.

(c) **Por qué podría ser mejor en algunos casos que un proceso se ejecute en el mismo procesador**: En algunos casos, es preferible que un proceso se ejecute en el mismo procesador debido a la afinidad de caché. Los procesadores modernos tienen múltiples niveles de caché, y cuando un proceso se ejecuta en el mismo procesador en el que se ejecutó anteriormente, es más probable que los datos necesarios estén ya en la caché del procesador, lo que puede mejorar el rendimiento y reducir la latencia.

(d) **Cambiar la afinidad de un proceso en Windows y en GNU/Linux**: Sí, tanto en Windows como en GNU/Linux, los usuarios con los permisos adecuados pueden cambiar la afinidad de un proceso. En Windows, puede hacerse a través del "Administrador de tareas" o mediante el comando `wmic process` en la línea de comandos. En GNU/Linux, se puede lograr con comandos como `taskset` o mediante herramientas de administración de tareas como `htop`.

(e) **Balanceo de carga (load balancing)**: El balanceo de carga es el proceso de distribuir equitativamente la carga de trabajo entre múltiples recursos, como procesadores, servidores o nodos de red, con el fin de optimizar el rendimiento y la utilización de recursos. Esto se hace para evitar cuellos de botella, mejorar la escalabilidad y garantizar un uso eficiente de los recursos.

(f) **Comparación entre afinidad y balanceo de carga y cómo afectan uno al otro**: La afinidad se refiere a la preferencia de un proceso por un procesador específico, lo que puede mejorar el rendimiento en términos de caché y latencia. El balanceo de carga, por otro lado, busca distribuir la carga de trabajo de manera uniforme para aprovechar al máximo todos los recursos disponibles.

Ambos conceptos pueden afectarse mutuamente. Por un lado, el balanceo de carga puede mover procesos entre procesadores para distribuir la carga de manera uniforme, lo que podría cambiar la afinidad que tenía un proceso. Por otro lado, una política de afinidad rígida podría dificultar el equilibrio de la carga, ya que limita la capacidad del sistema para asignar tareas de manera eficiente. En última instancia, la decisión de aplicar afinidad o balanceo de carga dependerá de los requisitos específicos del sistema y los objetivos de rendimiento.

25d) Obviamente la del inciso B pq cuando tenes ocupado un procesador usas el otro, te ahorras de esperar un ciclo. Tenes mejor Tiempo de retorno y de espera.