Práctica 4

1. Responda en forma sintética sobre los siguientes conceptos:

a) Programa y Proceso.

Programa la totalidad de instrucciones compiladas o de un lenguaje interpretado residentes en memoria secundaria y que hacen al comportamiento de una utilidad o aplicación, junto con los datos asociados. Un proceso, por otra parte, es una instancia de un programa en ejecución, un conjunto de instrucciones y datos, junto con las estructuras asociadas de control, con una tarea específica a cumplir y que residirá en memoria principal como tal hasta cumplirla (aunque puede ser enviado temporalmente a memoria secundaria).

b) Defina Tiempo de retorno (TR) y Tiempo de espera (TE) para un *job*.

Tiempo de Retorno: es el tiempo total que tarda un proceso determinado desde que es cargado efectivamente en memoria hasta que concluye su tarea en el CPU

Tiempo de Espera: es el tiempo de existencia de un proceso durante el cual no está haciendo uso del CPU. Es básicamente la diferencia entre el tiempo de retorno y el tiempo de ocupación del procesador

c) Defina Tiempo Promedio de Retorno (TPR) y Tiempo promedio de espera (TPE) para un lote de *jobs*.

TPR: es el promedio de tiempos de retorno entre todos los procesos pertenecientes al lote. Se calcula como la suma del TR de cada proceso, dividida entre la cantidad de procesos.

TPE: es el promedio de tiempos de espera calculado entre todos los procesos involucrados en el lote. Se calcula como la suma del TE de todos los procesos, dividida entre la cantidad de procesos.

d) ¿Qué es el Quantum?

Es el tiempo de procesador que se le asigna a los procesos en el algoritmo Round Robin. Determina la longitud máxima de la ráfaga de procesador que podrá utilizar cada proceso al entrar al CPU, puede ser más corta si es que el proceso concluye su tarea o debe esperar algún otro recurso antes de acabar la ráfaga.

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

Algoritmos **apropiativos** son aquellos en los que los procesos pueden ser sacados de la CPU según disparadores externos al propio proceso. En estos casos el SO fuerza la salida de un proceso del procesador para asignarle el uso a otro, independientemente de si el proceso retirado concluyó o no su uso del CPU.

Algoritmos **no apropiativos** son, por el contrario, aquellos en los que los procesos no abandonan el CPU más que por condiciones internas del proceso, como haber concluido su uso del procesador, o requerir la conclusión de una operación de E/S.

f) ¿Qué tareas realizan?

i. Short Term Scheduler es el que selecciona, de entre los procesos en estado *ready*, aquel que va a pasar a ser ejecutado.

ii. Long Term Scheduler se encarga de la admisión de procesos, es decir el paso de estado nuevo a listo.

iii. Medium Term Scheduler se ocupa del *swap out* de la memoria de el o los procesos que sea necesario para mantener el equilibrio del sistema. También se ocupa de la selección carga en memoria principal (*swap in*) de procesos puestos en este estado.

g) ¿Qué tareas realiza el Dispatcher? su tarea es el cambio de contexto entre procesos; la descarga de los registros del procesador y otros datos relevantes a las estructuras de datos que se utilizan para controlar el proceso (PCB) saliente, y la carga en el procesador y otros puntos de los mismos datos del proceso entrante.

2. Procesos:

(a) 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):

i. **`top`:** provee una vista dinámica y en tiempo real del sistema en ejecución. Puede tanto mostrar información del sistema en general, como una lista de tareas que están siendo administradas por el kernel. Los tipos de información mostrados en ambos espacios (información del sistema y de tareas) pueden configurarse hacerse persistentes entre reinicios del sistema
El programa incluye una interfaz interactiva limitada para la manipulación de procesos.

ii. **`htop`:** es un visualizador de procesos similar a top, pero permite recorrer la ventana de forma vertical y horizontal, de modo que se pueden ver todos los procesos ejecutándose en el sistema, junto con sus líneas de comando completas, de la misma forma que verlos como un árbol de procesos, seleccionando varios procesos y trabajado sobre ellos a la vez.
Las tareas relacionadas con los procesos (terminarlos, modificar su nice number), se pueden llevar a cabo sin indicar sus PIDs.

iii. **`ps`:** lista los procesos en ejecución

iv. **`pstree`:** muestra los procesos en ejecución como un árbol. El comando une ramas idénticas colocándolas entre corchetes y prefijando el contador de repeticiones (ejemplo: padre---3*[proceso]). Los hilos hijo se muestran bajo el proceso padre y se indican con el nombre del proceso padre entre llaves (ejemplo: padre---8*[{padre}]). Al final de la declaración de opciones puede indicarse un pid, en cuyo caso se tomará como proceso raíz del árbol, o un usuario, con lo que se mostrarán todos los árboles cuya raíz sea un proceso del que el usuario es propietario. Si no se indica ninguna de las opciones, se muestra un árbol con init o systemd como raíz.

v. **`kill`:** envía la señal especificada al o a los procesos indicados. Si no se especifica una señal, se envía TERM, que eliminará los procesos que no puedan capturarla. La sintaxis es ***kill [-s** señal* **| *-p] pid…*** o ***kill -l [**señal**]***
pid puede ser:

- Un número natural mayor a 0, en cuyo caso se señalará al proceso con ese PID
- 0: en cuyo caso todos los procesos en el grupo de procesos actual serán señalados
- -1: todos los procesos con PID mayor a uno serán señalados (salvo el propio *kill*)
- -n, siendo n un número mayor a 1: todos los procesos en el grupo de procesos n son señalados. Si se usa un argumento de este tipo, se debe especificar la señal primero, o se debe anteponer --, de otra forma -n será interpretada como la señal a enviar.

vi. **`pgrep`:** busca todos los procesos actualmente en el sistema cuyas características coincidan exactamente con las indicadas en la línea de comandos, y lista su PID en pantalla.
La sintaxis es **`pgrep [-flvx] [-d`** *`delimiter`*`] [-n|-o] [-P` *`ppid`*`,...] [-g` *`pgrp`*`,...] [-s` *`sid`*`,...] [-u` *`euid`*`,...] [-U` *`uid`*`,...] [-G` *`gid`*`,...] [-t` *`term`*`,...] [`*`pattern`*`]`
`-d` *`delimitador`*: establece la cadena que se utilizará para delimitar cada PID en la salida, por defecto es un salto de línea (exclusivo de **`pgrep`**).
`-f`: el patrón *`pattern`* normalmente sólo se compara con el nombre del proceso, si se establece **`-f`** se comparará con la línea de comandos completa.

-g *pgrp*,...**:** sólo tomará procesos que se correspondan con los PGIDs indicados. El PGID 0 es interpretado como el PGID bajo el que se ejecuta el propio comando.
-G *gid*,...**:** sólo tomará procesos cuyo ID real de grupo coincida con uno de los indicados. Se pueden utilizar valores numéricos o simbólicos.
-l: lista el nombre del proceso además de su PID. (exclusivo de **pgrep**).
-n: sólo selecciona el más nuevo de los procesos encontrados (el que haya iniciado más recientemente).
-o: el opuesto a -n, selecciona al más antiguo de los procesos encontrados.
-P *ppid*,...**:** sólo selecciona procesos cuyo PPID (*parent process ID*) coincida con uno de los indicados.
-s *sid*,...**:** sólo selecciona aquellos procesos cuyo ID de sesión de proceso coincide con uno de los indicados. El ID de sesión 0 es interpretado como aquel bajo el que se ejecuta el propio comando.
-t *term*,...**:** sólo selecciona procesos cuya terminal de control se encuentra entre las indicadas. El nombre de la terminal debe ser especificado con el prefijo "/dev/".
-u *euid*,...**:** sólo selecciona procesos cuyo ID de usuario efectivo coincide con el indicado. Pueden utilizarse tanto valores simbólicos como numéricos.
-U *uid*,...**:** sólo selecciona procesos cuyo ID de usuario real coincide con el indicado. Pueden utilizarse tanto valores simbólicos como numéricos.
-v: realiza la negación de la comparación (selecciona los que no respetan el/los patrones)
-x: sólo selecciona aquellos procesos cuyo nombre coincide exactamente con el patrón indicado (o su línea de comando de origen si se selecciona la opción **-f**)

vii. **pkill:** busca todos los procesos cuyas características coincidan exactamente con las indicadas en la línea de comandos, y les envía el mensaje indicado en el llamado al comando. Si no se indica una señal, se enviará la predeterminada SIGTERM.
La sintaxis es **pgrep [***-signal***] [-fvx] [-n|-o] [-P** *ppid*,...**] [-g** *pgrp*,...**] [-s** *sid*,...**] [-u** *euid*,...**] [-U** *uid*,...**] [-G** *gid*,...**] [-t** *term*,...**] [***pattern***]**
Comparte las opciones con el comando **pgrep**, con la excepción de:
*-signal***:** define la señal a ser enviada a cada uno de los procesos que coincidan con los patrones indicados. Pueden utilizarse indistintamente valores simbólicos o numéricos de señal.
(y aquellas indicadas en **pgrep** como exclusivas de dicho comando)

viii. **killall:** envía la señal especificada al o a los proceso indicados por nombre. Si no se especifica una señal, se envía TERM, que eliminará los procesos que no puedan capturarla. La sintaxis es ***kill [-s** señal**| -p] nombre…*** o ***kill -l [**señal**]***. Si el nombre de comando indicado no es una expresión regular (opción -r) y contiene /, el proceso ejecutando ese archivo será seleccionado para morir independientemente de su nombre.

ix. **renice:** modifica la prioridad de uno o más procesos en ejecución. El primer argumento es el valor de prioridad (*nice number*) a ser utilizado. Los demás argumentos son identificados como PIDs (la opción por defecto), PGIDs, UIDs o nombres de usuario. Ejecutar **renice** sobre un grupo de procesos (por PGID, UID o nombre de usuario) cambia la prioridad de planificación de todo el grupo involucrado.
La sintaxis es **renice** [**-n**] *priority* [**-g**|**-p**|**-u**] *identifier...*
Opciones:
-n, **--priority** *priority***:** especifica la prioridad a ser asignada al proceso, grupo de procesos o usuario. Indicar **-n** o **--priority** es opcional, pero si se lleva a cabo debe ser el primer argumento
-g, **--pgrp** | **-p**, **--pid** | **-u**, **--user:** aplica la prioridad indicada a los procesos cuyo PGID, PID (la opción por defecto), o usuario (indicado con nombre o UID) coincida con los listados a continuación de la opción.

x. **xkill:** fuerza la desconexión de un X server con uno de sus clientes (básicamente permite cerrar ventanas de programas). Forzar la desconexión no implica necesariamente que los procesos asociados al cliente terminarán de forma correcta, o siquiera que serán terminados.
La sintaxis es **xkill** [-display *displayname*] [-id *resource*] [-button number] [-frame] [-all]
Si no se indica ninguna de las opciones, se mostrará un cursor en forma de x (o una calavera, según la distribución y conjunto de símbolos del sistema) con el que se podrá seleccionar el cliente (ventana) a cerrar.

xi. **atop**: es un monitor interactivo de la carga de trabajo en un sistema operativo Linux. Indica la ocupación de los recursos más críticos de hardware - desde una perspectiva de rendimiento - a nivel de sistema (i.e. CPU, memoria, disco y red). También muestra qué procesos son responsables de la carga indicada de cpu y memoria. La carga de disco y red dependen de la presencia de configuraciones y módulos del kernel.
Cada un determinado intervalo (predeterminado: 10 segundos, pero se puede modificar en la línea de comandos) se muestra información sobre la ocupación de recursos del sistema, seguida por una lista de procesos que han estado activos durante el último intervalo (si algún proceso no fue modificado durante este intervalo, no será mostrado, salvo que se haya presionado la tecla 'a'). Si la lista de procesos activos no entra en la pantalla, se mostrará todos los que entren, ordenados en función de su actividad.
Al igual que top, los intervalos se repiten hasta alcanzar el número de muestras indicadas en el comando de llamada al programa, o hasta que se presione la tecla 'q' en caso de estar en modo interactivo.
Cuando se inicia, el programa evalúa si la salida estándar está conectada con una pantalla con un archivo o pipe. En el primer caso produce códigos de control de pantalla, y se comporta de forma interactiva; en el segundo caso produce una salida de texto plano en formato ASCII.
En modo interactivo, la salida de atop es escalada de forma dinámica en función de las dimensiones actuales de la ventana o pantalla. Si se modifica el ancho de la pantalla o ventana, se agregarán o quitarán columnas de manera automática, mostrando aquellas más importantes que entren en la dimensión actual (para este fin, cada columna tiene asignado un valor de relevancia)

(b) 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 ;
}
```

i. ¿Cuántas líneas con la palabra "Proceso" aparecen al final de la ejecución de este programa? 8

ii. ¿El número de líneas es el número de procesos que han estado en ejecución?. Ejecute el programa y compruebe si su respuesta es correcta, Modifique el valor del bucle for y compruebe los nuevos resultados.

(c) 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 ;
pid_t pid ;
printf ( "Comienzo . : \ n " ) ;
for ( c=0 ; c<3 ; c++ )
{
   pid = fork ( ) ;
}
p++
printf ( " Proceso %d\n " ) ;
return 0 ;
}
```

i. ¿Qué valores se muestran por consola?
Proceso 1*8

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

iii. ¿Cuál es el valor (o valores) que aparece?. Ejecute el programa y compruebe si su respuesta es correcta, Modifique el valor del bucle for y el lugar dónde se incrementa la variable p y compruebe los nuevos resultados.

(d) Comunicación entre procesos:
i. Investigue la forma de comunicación entre procesos a través de pipes.
Un pipe o canal es un búfer circular que permite que dos procesos se comuniquen entre sí siguiendo el modelo productor-consumidor. En sentido de implementación, es un seudo archivo que reemplaza la salida estándar de un proceso y la entrada estándar de otro, de forma tal que puedan enviar y recibir mensajes respectivamente. Funciona como una cola de tipo FIFO, en la que las sucesivas escrituras se van leyendo por orden de llegada.
Cuando se crea una tubería, se le establece un tamaño fijo en bytes. Cuando un proceso intenta escribir en la tubería, la petición de escritura se ejecuta inmediatamente si hay suficiente espacio; en caso contrario, el proceso se bloquea. De manera similar, un proceso que lee se bloquea si intenta leer más bytes de los que están actualmente en la tubería; en caso contrario, la petición de lectura se ejecuta inmediatamente. El sistema operativo asegura la exclusión mutua: es decir, en cada momento sólo puede acceder a una tubería un único proceso.
Hay dos tipos de tuberías: con nombre y sin nombre. Sólo los procesos relacionados pueden compartir tuberías sin nombre, mientras que los procesos pueden compartir tuberías con nombre tanto si

están relacionados como si no.
Por último, un proceso puede enviar una señal de error POSIX `SIGPIPE`, que indica que recibió un pipe que no tiene lectores.

ii. ¿Cómo se crea un pipe en C?.
A través de la instrucción `pipe(&fd[N])` con `N = 0` o `N = 1` y `fd` un arreglo de dos enteros

iii. ¿Qué parámetro es necesario para la creación de un pipe?. Explique para qué se utiliza.
Se debe establecer como parámetro un arreglo de dos enteros, indicando el primer dato ([0]) si el proceso recibirá datos a través del canal (es decir que se convierte en el descriptor de archivo de su entrada estándar), y el segundo dato si utilizará el pipe para escribir (convirtiéndolo en el descriptor de archivo de su salida estándar).
Para lograr una comunicación correcta, los procesos involucrados deberán cerrar el extremo del canal que no les incube (`close(&fd[0])` si el pipe fue abierto con `pipe(&fd[1])` y viceversa. De otra forma, el pipe nunca se dará por cerrado, puesto que no aparecerá una marca de `EOF`.

iv. ¿Qué tipo de comunicación es posible con pipes?
Los canales o tuberías sólo permiten comunicación de una sóla vía, es decir que un proceso sólo puede escribir o leer un pipe, pero no ambas a la vez.

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

(f) ¿Qué significa que un proceso sea “CPU Bound” y “I/O Bound”?
***CPU Bound*:** significa que el proceso requiere principalmente de tiempo de procesador para su ejecución
***I/O Bound*:** significa que el proceso depende en gran medida de operaciones de E/S para su ejecución

(g) ¿Cuáles son los estados posibles por los que puede atravesar un proceso?
- **Nuevo (new):** implica la carga de datos del proceso en memoria, incluyendo las estructuras asociadas (PCB, etc.). Esta etapa se considera concluida una vez que termina de cargarse en memoria, y se considera admitido por el sistema.
- **Listo para ejecutar (ready/ready to run):** según las prioridades establecidas para el proceso, deberá o no esperar a que su primera instrucción sea cargada en el CPU. En este estado el proceso compite con otros por el uso del procesador.
- **Ejecutándose (running):** es la ejecución normal de las instrucciones involucradas en el proceso (luego de haber sucedido una selección y el correspondiente *context switch*). De este estado pueden suceder tres caminos: por un lado darse la finalización normal de sus instrucciones; otra opción es que sea interrumpido, retornando al estado *ready*.
- **Esperando su ejecución/bloqueado (waiting):** si el proceso requiere uso de algún recurso E/S, o por algún motivo se hace esperar, hasta que el tiempo necesario no haya transcurrido., las instrucciones subsiguientes a él no podrán ejecutarse. Por este motivo el proceso es puesto en espera, sin posibilidad de ser cargado en el CPU. Una vez concluida la E/S o pasado el tiempo necesario para que se pueda retomar el proceso, este es cargado nuevamente en la cola de ejecuciones pasando al estado *ready*.
- **Finalizado/saliente (terminated):** es más que nada la eliminación de memoria principal de todas las estructuras asociadas, finaliza con el borrado de la PCB.

- **Suspendido:** es un estado que puede superponerse con listo y bloqueado, implica que el proceso en cuestión, más allá de su estado en cuanto a su capacidad intrínseca de acceder al procesador, no se encuentra cargado en memoria principal, si no exclusivamente en memoria secundaria.

(h) Explique mediante un diagrama las posibles transiciones entre los estados.

(i) ¿Que scheduler de los mencionados en 1 f se encarga de las transiciones?
De las transiciones del estado listo a ejecutándose, se encarga el planificador de corto plazo
De las transiciones a y desde el estado suspendido se encarga el planificador de mediano plazo
De la admisión y transición al estado listo se encarga el planificador de largo plazo

3. Para los siguientes algoritmos de *scheduling*:

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

(a) Explique su funcionamiento mediante un ejemplo.
Suponiendo los siguientes procesos:

PID	Orden	CPU requerido	Prioridad*
1	0	4	2
2	1	5	1
3	2	2	1

**éste dato sólo es relevante para el algoritmo de prioridades*

PID: es el ID de cada proceso
Orden: el orden en el cual los procesos llegan a estado listo
CPU requerido: el tiempo de CPU que cada proceso requiere para completarse
Prioridad: el valor de prioridad del proceso, un valor más bajo implica mayor prioridad.

FCFS: ejecutará los procesos por orden de llegada, permitiendo que se expulsen del CPU automáticamente al terminarse (o requerir otro recurso). La historia del CPU sería:

PID	0	1	2	3	4	5	6	7	8	9	10
1	>			<							
2					>				<		
3										>	<

Como puede verse, el comportamiento es bastante predecible, en caso de que un proceso (supongamos el 1) dependiera de una operación I/O (supongamos en el instante 1 de CPU), éste hubiera salido del procesador, siendo reasignado el mismo al siguiente proceso en cola, y el proceso 1 hubiera sido reencolado una vez terminada la operación I/O, resultando en una historia de CPU como la siguiente (asumiendo que la operación I/O finaliza antes que el proceso 3):

PID	0	1	2	3	4	5	6	7	8	9	10
1	>										<

2		**>**				**<**					
3							**>**	**<**			

SJF: irá tomando, de los procesos listos para ejecutarse, aquél que menor tiempo de ejecución consuma
Round Robin:
Prioridades:

(b) ¿Alguno de ellos requiere algún parámetro para su funcionamiento?
El *round robin* requiere la definición de medida del quantum

(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?
En el caso de quantum variable, cada proceso debe almacenar la información de su quantum, mientras que en el caso de timer fijo, se debe almacenar en una estructura o espacio accesible directamente por el SO.

5. Se tiene el siguiente lote de procesos que arriban al sistema en el instante 0 (cero):

JOB	Unidades de CPU
1	7
2	15
3	12
4	4
5	9

(a) Realice los diagramas de Gantt según los siguientes algoritmos de scheduling:
i. FCFS (First Come, First Served)
ii. SJF (Shortest Job First)
iii. Round Robin con quantum = 4 y Timer Fijo
iv. Round Robin con quantum = 4 y Timer Variable

(b) Para cada algoritmo calcule el TR y TE para cada job así como el TPR y el TPE.

(c) En base a los tiempos calculados compare los diferentes algoritmos.
Los algoritmos no apropiativos en este caso dan mucho mejores resultados en promedio, aventajando en particular el SJF a los demás. Sin embargo, la disparidad en la atención recibida por los procesos es mucho mayor en términos proporcionales en los casos no apropiativos que en los apropiativos (lo que es esperable, puesto que es una de sus principales diferencias teóricas).

6. Se tiene el siguiente lote de procesos:

JOB	Llegada	Unidades de CPU
1	0	4
2	2	6
3	3	4
4	6	5
5	8	2

(a) Realice los diagramas de Gantt según los siguientes algoritmos de scheduling:
i. FCFS (First Come, First Served)
ii. SJF (Shortest Job First)
iii. Round Robin con quantum = 1 y Timer Variable
iv. Round Robin con quantum = 6 y Timer Variable

(b) Para cada algoritmo calcule el TR y TE para cada job así como el TPR y el TPE.

(c) En base a los tiempos calculados compare los diferentes algoritmos.
El algoritmo SJF es el más eficiente en el caso promedio, mientras que el FCFS y el RR de quantum 6 no presentan una gran diferencia entre sí (ni tanta respecto al SJF). Claramente menos eficiente resulta el RR de quantum 1

(d) En el algoritmo Round Robin, qué conclusión se puede sacar con respecto al valor del quantum.
Evidentemente un valor de quantum muy bajo puede conllevar un retraso importante en la atención de procesos (tanto su TR como TE), y esto sin contar los retrasos que se puede suponer implicarían los constantes cambios de contexto que esto involucra.

(e) ¿Para el algoritmo Round Robin, en qué casos utilizaría un valor de quantum alto y que ventajas y desventajas obtendría?
Lo utilizaría en aquellos casos en los que los procesos a atender requieren en general bastante tiempo de CPU (una medida similar o superior y proporcional al quantum). La ventaja principal obtenida sería la atención relativamente pareja de los procesos, y eficiente en relación con este primer principio. La desventaja es clara en cuanto el tiempo de CPU requerido por cada proceso es muy dispar y no proporcional al quantum, en ese caso habrá procesos esperando varios ciclos para poder ocupar el CPU sólo por una fracción de quantum para poder darse por terminados, lo que en caso de un timer variable

además implicaría que los procesos inmediatamente subsiguientes a estos eventos obtendrán sólo la fracción restante del quantum, generando desfases aún mayores.

7. Una variante al algoritmo SJF es el algoritmo SJF apropiativo o SRTF (*Shortest Remaining Time First*):

(a) Realice el diagrama de Gantt para este algoritmo según el lote de trabajos del ejercicio 6.

(b) ¿Nota alguna ventaja frente a otros algoritmos?
Parece ser bastante más eficiente, con especial ganancia en lo que a tiempo de espera se refiere.

8. Suponga que se agregan las siguientes prioridades al lote de procesos del ejercicio 6, donde un menor número indica mayor prioridad:

JOB	Prioridad
1	3
2	4
3	2
4	1
5	2

(a) Realice el diagrama de Gantt correspondiente al algoritmo de planificación por prioridades según las variantes:
i. No Apropiativa
ii. Apropiativa

(b) Calcule el TR y TE para cada job así como el TPR y el TPE.

(c) ¿Nota alguna ventaja frente a otros algoritmos? Bajo qué circunstancias lo utilizaría y ante qué situaciones considera que la implementación de prioridades podría no ser de mayor relevancia?
Es de similar eficiencia al SJF (de hecho la variante no apropiativa es exactamente igual). La aplicación de prioridades podría ser útil en tanto y en cuanto estas puedan parecer relativamente inversas al tiempo requerido de CPU, de otra forma, un algoritmo SJF o SJRF cumplirían exactamente la misma función.

9. Inanición (Starvation)

(a) ¿Qué significa?
La inanición en términos generales es la situación en la que un determinado proceso no puede completarse por falta de recursos. En el contexto específico de algoritmos de planificación, se habla de inanición de un proceso cuando éste no puede avanzar debido a que no se le asigna tiempo de CPU.
(b) ¿Cuál/es de los algoritmos vistos puede provocarla?
Cualquier algoritmo que involucre algún tipo de diferenciación entre procesos y asigne recursos con base en ese criterio es susceptible de generar inanición, traducido a tipos de algoritmos, implica que los tipos SJF, SRTF y por prioridades, todos pueden generar inanición; los primeros dos en aquellos procesos que sean más largos que, y el tercero en aquellos procesos de menor prioridad.

(c) ¿Existe alguna técnica que evite la inanición para el/los algoritmos mencionados en b?
En los tres casos se puede aplicar una técnica que involucre forzar la entrada al procesador de procesos que lleven más de un determinado tiempo esperando que se les asigne dicho recurso. En el caso de algoritmos por prioridades, en general la técnica consiste en aumentar la prioridad de los procesos en la medida que aumenta su tiempo de espera, de modo que en algún momento alcancen un nivel de prioridad que lleve a que se les asigne espacio en el CPU.

10. Los procesos, durante su ciclo de vida, pueden realizar operaciones de I/O como lecturas o escrituras a disco, cintas, uso de impresoras, etc.

El SO mantiene para cada dispositivo que se tiene en el equipo, una cola de procesos que espera por la utilización del mismo (al igual que ocurre con la Cola de Listos y la CPU, ya que la CPU es un dispositivo mas). Cuando un proceso en ejecución realiza una operación de I/O el mismo es expulsado de la CPU y colocado en la cola correspondiente al dispositivo involucrado en la operación.
El SO dispone también de un "I/O Scheduling" que administra cada cola de dispositivo a través de algún algoritmo (FCFS, Prioridades, etc.). Si al colocarse un proceso en la cola del dispositivo, la misma se encuentra vacía el mismo será atendido de manera inmediata, caso contrario, deberá esperar a que el SO lo seleccione según el algoritmo de scheduling establecido.
Los mecanismos de I/O utilizados hoy en día permiten que la CPU no sea utilizada durante la operación, por lo que el SO puede ejecutar otro proceso que se encuentre en espera una vez que el proceso bloqueado por la I/O se coloca en la cola correspondiente.
Cuando el proceso finaliza la operación de I/O el mismo retorna a la cola de listos para competir nuevamente por la utilización de la CPU.

Para los siguientes algoritmos de Scheduling:

- FCFS
- Round Robin con quantum = 2 y timer variable.

Y suponiendo que la cola de listos de todos los dispositivos se administra mediante FCFS, realice los diagramas de Gantt según las siguientes situaciones:

(a) Suponga que al lote de procesos del ejercicio 6 se agregan las siguientes operaciones de entrada salida:

JOB	I/O (rec, ins, dur)
1	(R1, 2, 1)
2	(R2, 3, 1) (R2, 5, 2)
4	(R3, 1, 2) (R3, 3, 1)

(b) Suponga que al lote de procesos del ejercicio 6 se agregan las siguientes operaciones de entrada salida:

JOB	I/O (rec, ins, dur)
1	(R1, 2, 3) (R1, 3, 2)
2	(R2, 3, 2)

3	(R2, 2, 3)
4	(R1, 1, 2)

11. Algunos algoritmos pueden presentar ciertas desventajas cuando en el sistema se cuenta con procesos ligados a CPU y procesos ligados a entrada salida. Analice las mismas para los siguientes algoritmos:

(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.

(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. Para equiparar la desventaja planteada en el ejercicio 11), se plantea la siguiente modificación al algoritmo:
Algoritmo VRR (*Virtual Round Robin*): Este algoritmo funciona igual que el Round Robin, con la diferencia que cuando un proceso regresa de una I/O se coloca en una cola auxiliar. Cuando se tiene que tomar el próximo proceso a ejecutar, los procesos que se encuentran en la cola auxiliar tienen prioridad sobre los otros. Cuando se elije un proceso de la cola auxiliar se le otorga el procesador por tantas unidades de tiempo como le falto ejecutar en su ráfaga de CPU anterior, esto es, se le otorga la CPU por un tiempo que surge entre la diferencia del quantum original y el tiempo usado en la última ráfaga de CPU.

(a) Analice el funcionamiento de este algoritmo mediante un ejemplo. Marque en cada instante en que cola se encuentran los procesos.

(b) Realice el ejercicio 10)a) nuevamente considerando este algoritmo, con un quantum de 2 unidades y Timer Variable.

13. Suponga que un SO utiliza un algoritmo de VRR con Timer Variable para planificar sus procesos. Para ello, el quantum es representado por un contador, que es decrementado en 1 unidad cada vez que ocurre una interrupción de reloj. ¿Bajo este esquema, puede suceder que el quantum de un proceso nunca llegue a 0 (cero)? Justifique su respuesta.

14. El algoritmo SJF (y SRTF) tiene como problema su implementación, dada la dificultad de conocer la duración de la próxima ráfaga de CPU. Es posible realizar una

estimación de la próxima, utilizando la media de las ráfagas de CPU para cada proceso.
Así, por ejemplo, podemos tener la siguiente fórmula:

$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$

Donde:

T_i = duración de la ráfaga de CPU i-ésima del proceso.
S_i = valor estimado para el i-ésimo caso
S_1 = valor estimado para la primer ráfaga de CPU. No es calculado.

(a) Suponga un proceso cuyas ráfagas de CPU reales tienen como duración: 6, 4, 6, 4, 13, 13, 13 Calcule qué valores se obtendrían como estimación para las ráfagas de CPU del proceso si se utiliza la fórmula 1, con un valor inicial estimado de S_1=10.

Ráfaga (n)	1	2	3	4	5	6	7	8
Estimación (S_n)	*10*	*6*	*5*	*5.3*	*5*	*6.6*	*7.6*	*8.4*
Duración real (T_n)	6	4	6	4	13	13	13	

La fórmula anterior 1 le da el mismo peso a todos los casos (siempre calcula la media). Es posible reescribir la fórmula permitiendo darle un peso mayor a los casos más recientes y menor a casos viejos (o viceversa). Se plantea la siguiente fórmula:

$S_{n+1} = \alpha T_n + (1 - \alpha) S_n$

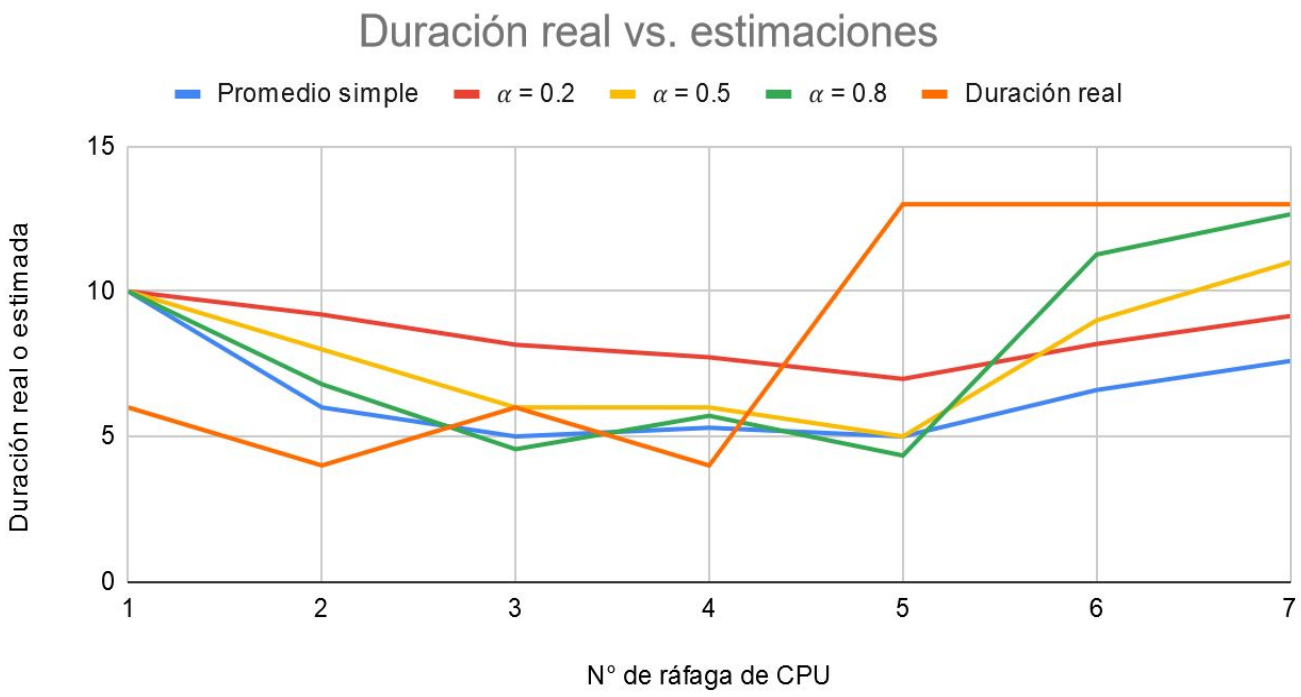
Con $0 < \alpha < 1$.

(b) Analice para que valores de α se tienen en cuenta los casos más recientes.
Para valores de $\alpha \rightarrow 1$. En ese caso tiene más peso el primer término de la ecuación, que representa el tiempo más reciente.

(c) Para la situación planteada en a) calcule qué valores se obtendrían si se utiliza la fórmula 2 con
α = 0, 2
α = 0, 5
α = 0, 8

	Ráfaga (n)								
	α	1	2	3	4	5	6	7	8
Estimación (Sn)	0.2	*10*	*9.20*	*8.16*	*7.73*	*6.98*	*8.19*	*9.15*	*9.92*
	0.5	*10*	*8.00*	*6.00*	*6.00*	*5.00*	*9.00*	*11.00*	*12.00*
	0.8	*10*	*6.80*	*4.56*	*5.71*	*4.34*	*11.27*	*12.65*	*12.93*
Duración real (Tn)		6	4	6	4	13	13	13	

(d) Para todas las estimaciones realizadas en a y c ¿Cuál es la que más se asemeja a las ráfagas de CPU reales del proceso?



15. Colas Multinivel

Hoy en día los algoritmos de planificación vistos se han ido combinando para formar algoritmos más eficientes. Así surge el algoritmo de Colas Multinivel, donde la cola de procesos listos es divida en varias colas, teniendo cada una su propio algoritmo de planificación.

(a) Suponga que se tienen dos tipos de procesos: Interactivos y Batch. Cada uno de estos procesos se coloca en una cola según su tipo. ¿Qué algoritmo de los vistos utilizaría para administrar cada una de estas colas?
Virtual Round Robin para la cola de procesos interactivos
FCFS para la cola de procesos *batch*

A su vez, se utiliza un algoritmo para administrar cada cola que se crea. Así, por ejemplo, el algoritmo podría determinar mediante prioridades sobre qué cola elegir un proceso.

(b) Para el caso de las dos colas vistas en a: ¿Qué algoritmo utilizaría para planificarlas?
SRTF con una política de *aging*

16. Suponga que en un SO se utiliza un algoritmo de planificación de colas multinivel.

El mismo cuenta con 3 colas de procesos listos, en las que los procesos se encolan en una u otra según su prioridad. Hay 3 prioridades (1 , 2 , 3), donde un menor número indica mayor prioridad.
Se utiliza el algoritmo de prioridades para la administración entre las colas.
Se tiene el siguiente lote de procesos a ser procesados con sus respectivas operaciones de I/O:

Job	Llegada	CPU	I/O (rec, ins, dur)	Prioridad
1	0	9	(R1, 4, 2) (R2, 6, 3) (R1, 8, 3)	1
2	1	5	(R3, 3, 2) (R3, 4, 2)	2
3	2	5	(R1, 4, 1)	3
4	3	7	(R2, 1, 2) (R2, 5, 3)	2
5	5	5	(R1, 2, 3) (R3, 4, 3)	1

Suponiendo que las colas de cada dispositivo se administran a través de FCFS y que cada cola de procesos listos se administra por medio de un algoritmo RR con un quantum de 3 unidades y Timer Variable, realice un diagrama de Gantt:

(a) Asumiendo que NO hay apropiación entre los procesos

(b) Asumiendo que hay apropiación entre los procesos

17. En el esquema de Colas Multinivel, cuando se utiliza un algoritmo de prioridades para administrar las diferentes colas los procesos pueden sufrir starvation. La técnica de envejecimiento se puede aplicar a este esquema, haciendo que un proceso cambie de una cola de menor prioridad a una de mayor prioridad, después de cierto periodo de tiempo que el mismo se encuentra esperando en su cola. Luego de llegar a una cola en la que el proceso llega a ser atendido, el mismo retorna a su cola original. Por ejemplo: Un proceso con prioridad 3 está en cola su cola correspondiente. Luego de X unidades de tiempo, el proceso se mueve a la cola de prioridad 2. Si en esta cola es atendido, retorna a su cola original, en caso contrario luego de sucederse otras X unidades de tiempo el proceso de mueve a la cola de prioridad 1. Esta última acción se repite hasta que el proceso obtiene la CPU, situación que hace que el mismo vuelva a su cola original.

(a) Para los casos a y b del ejercicio 16 realice el diagrama de Gantt considerando además que se tiene un envejecimiento de 4 unidades.

18. La situación planteada en el ejercicio 17, donde un proceso puede cambiar de una cola a otra, se la conoce como Colas Multinivel con Realimentación.

Suponga que se quiere implementar un algoritmo de planificación que tenga en cuenta el tiempo de ejecución consumido por el proceso, penalizando a los que más tiempo de ejecución tienen. (Similar a la tarea del algoritmo SJF que tiene en cuenta el tiempo de ejecución que resta).
Utilizando los conceptos vistos de Colas Multinivel con Realimentación indique que colas implementaría, que algoritmo usaría para cada una de ellas así como para la administración de las colas entre sí. Tenga en cuenta que los procesos no deben sufrir inanición.

La solución podría consistir en la implementación de un turno rotatorio entre colas de prioridad. Cada vez que un proceso salga del procesador, entraría en la cola de prioridad inmediata inferior a aquella de la salió para entrar al CPU. Es decir, un proceso dado entra a la cola 0, y una vez atendido por el procesador, si es expulsado (sea porque pasa a estado bloqueado o porque se le agotó la rodaja de tiempo asignada), cuando esté listo lo estará en la cola 1, luego de su segunda atención pasará a la cola 2, etc. Cada una de estas colas no es atendida hasta que no estén vacías las colas de prioridad superior (lo que en UNIX se traduce en valores más próximos a 0), y los procesos en cada una de ellas son atendidos con planificación FCFS, salvo la última cola (ya que no hay una cola de menor prioridad a la

cual degradar los procesos) que se podría planificar con RR..
Sin más consideraciones, el método es muy susceptible de generar inanición en procesos largos. Para solucionar esto se pueden combinar dos estrategias:
- La asignación de rodajas de tiempo progresivamente más largas a las colas en la medida en que su prioridad disminuye. Así, si bien los procesos de baja prioridad pueden estar mucho tiempo esperando al procesador, esta penalización se puede ver compensada con la posibilidad de usarlo más tiempo una vez que entran. No se trata de una solución total al problema de inanición, ya que si las colas de mayor prioridad se mantienen ocupadas, de todas formas los procesos de menor prioridad no accederán al procesador.
- La implementación de algún mecanismo de aging: podría implementarse un mecanismo de envejecimiento, que aumente la prioridad de cada proceso en función del tiempo que lleva esperando en cada cola (de la misma manera que se explica en el ejercicio anterior). De esta forma sí se evita por completo la inanición de los procesos, ya que tarde o temprano podrán acceder al CPU, aunque de todas formas los procesos más largos se verán seriamente penalizados.

19. Un caso real: “Unix Clasico “ (SVR3 y BSD 4.3)

Estos sistemas estaban dirigidos principalmente a entornos interactivos de tiempo compartido. El algoritmo de planificación estaba diseñado para ofrecer buen tiempo de respuesta a usuarios interactivos y asegurar que los trabajos de menor prioridad (en segundo plano) no sufrieran inanición.
La planificación tradicional usaba el concepto de colas multinivel con realimentación, utilizando RR para cada uno de las colas y realizando el cambio de proceso cada un segundo (quantum). La prioridad de cada proceso se calcula en función de la clase de proceso y de su historial de ejecución. Para ello se aplican las siguientes funciones:
$CPUj(i) = CPUj(i - 1)\ 2$ (3)
$Pj(i) = Basej + CPUj(i)\ 2 + nicej$ (4)
Donde:
$CPUj(i)$ = Media de la utilización de la CPU del proceso j en el intervalo i.
$Pj(i)$ = Prioridad del proceso j al principio del intervalo i (los valores inferiores indican prioridad más alta).
$Basej$ = Prioridad base del proceso j.
$Nicej$ = Factor de ajuste.
La prioridad del proceso se calcula cada segundo y se toma una nueva decisión de planificación. El propósito de la prioridad base es dividir los procesos en bandas fijas de prioridad. Los valores de CPU y nice están restringidos para impedir que un proceso salga de la banda que tiene asignada. Las bandas definidas, en orden decreciente de prioridad, son:
Intercambio
Control de Dispositivos de I/O por bloques
Gestión de archivos
Control de Dispositivos de I/O de caracteres
Procesos de usuarios

Veamos un ejemplo: Supongamos 3 procesos creados en el mismo instante y con prioridad base 60 y un valor nice de 0. El reloj interrumpe al sistema 60 veces por segundo e incrementa un contador para el proceso en ejecución. Los sectores en celeste representan el proceso en ejecución.

(a) Analizando la jerarquía descrita para las bandas de prioridades: ¿Que tipo de actividad considera que tendrá más prioridad? ¿Por qué piensa que el scheduler prioriza estas actividades?
Tendrán más prioridad los procesos asociados con lecturas/escrituras en memoria secundaria y el

manejo de archivos. Priorizar estas tareas permite que aprovechen al máximo el poco uso de CPU que puedan necesitar, que se compensa con las relativamente largas esperas que puede implicar la necesidad de un recurso de memoria secundaria (comparativamente respecto al CPU). Por otro lado, el área de intercambio y la memoria secundaria son relevantes para mantener el propio control de procesos (uso de memoria virtual, respuesta a fallos de página, etc.).

(b) Para el caso de los procesos de usuarios, y analizando las funciones antes descritas: ¿Qué tipo de procesos se encarga de penalizar? (o equivalentemente se favorecen). Justifique
Se verán favorecidos los procesos vinculados a E/S, ya que éstos por lo general hacen menos uso del procesador en un momento determinado con lo que su valor para CPUj será menor, y con ello también su prioridad para un mismo valor de base y ajuste.

(c) La utilización de RR dentro de cada cola: ¿Verdaderamente favorece al sistema de Tiempo Compartido? Justifique.
Al menos puede considerarse que hace honor al nombre de la metodología, ya que efectivamente fuerza el uso compartido del procesador. Por otro lado, puede considerarse que balancea parcialmente la prioridad que reciben los procesos de E/S, ya que el RR específicamente penaliza a este tipo de procesos.

20. A cuáles de los siguientes tipos de trabajos:

(a) cortos acotados por CPU
(b) cortos acotados por E/S
(c) largos acotados por CPU
(d) largos acotados por E/S

benefician las siguientes estrategias de administración:

(a) prioridad determinada estáticamente con el método del más corto primero (SJF).
cortos acotados por CPU: particularmente beneficiados
cortos acotados por E/S: beneficiados en tanto y en cuanto no suelan entrar procesos largos
largos acotados por CPU: penalizados
largos acotados por E/S: altamente penalizados

(b) prioridad dinámica inversamente proporcional al tiempo transcurrido desde la última operación de E/S.
cortos acotados por CPU: penalizados
cortos acotados por E/S: particularmente beneficiados
largos acotados por CPU: altamente penalizados
largos acotados por E/S: beneficiados

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

Porque a partir de cierto punto, el quantum será más largo que el tiempo requerido por la mayoría de los procesos en espera, con lo que cada proceso que accedería al procesador lo abandonaría por cuenta propia (por haberse concluído o por bloquearse), y lo mismo sucedería con los procesos subsiguientes. 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).

22. Los sistemas multiprocesador pueden clasificarse en:

Homogéneos: Los procesadores son iguales. Ningún procesador tiene ventaja física sobre el resto.
Heterogéneos: Cada procesador tiene su propia cola y algoritmo de planificación.

Otra clasificación posible puede ser:

Multiprocesador débilmente acoplados: Cada procesador tiene su propia memoria principal y canales.
Procesadores especializados: Existen uno o más procesadores principales de propósito general y varios especializados controlados por el primero (ejemplo procesadores de E/S, procesadores Java, procesadores Criptográficos, etc.).
Multiprocesador fuertemente acoplado: Consta de un conjunto de procesadores que comparten una memoria principal y se encuentran bajo el control de un Sistema Operativo

(a) ¿Con cuál/es de estas clasificaciones asocia a las PCs de escritorio habituales?
Homogéneos, fuertemente acoplados

(b) ¿Qué significa que la asignación de procesos se realice de manera simétrica?*
En un multiprocesador simétrico, el núcleo puede ejecutar en cualquier procesador, y normalmente cada procesador realiza su propia planificación del conjunto disponible de procesos e hilos. El núcleo puede construirse como múltiples procesos o múltiples hilos, permitiéndose la ejecución de partes del núcleo en paralelo. Éste complica al sistema operativo, ya que debe asegurar que dos procesadores no seleccionan un mismo proceso y que no se pierde ningún proceso de la cola. Se deben emplear técnicas para resolver y sincronizar el uso de los recursos.

(c) ¿Qué significa que se trabaje bajo un esquema Maestro/esclavo?*
Con la arquitectura maestro/esclavo, el núcleo del sistema operativo siempre ejecuta en un determinado procesador. El resto de los procesadores sólo podrán ejecutar programas de usuario y, a lo mejor, utilidades del sistema operativo. El maestro es responsable de la planificación de procesos e hilos. Una vez que un proceso/hilo está activado, si el esclavo necesita servicios (por ejemplo, una llamada de E/S), debe enviar una petición al maestro y esperar a que se realice el servicio. Este enfoque es bastante sencillo y requiere pocas mejoras respecto a un sistema operativo multiprogramado uniprocesador. La resolución de conflictos se simplifica porque un procesador tiene el control de toda la memoria y recursos de E/S. Las desventajas de este enfoque son las siguientes:

- Un fallo en el maestro echa abajo todo el sistema.
- El maestro puede convertirse en un cuello de botella desde el punto de vista del rendimiento, ya que es el único responsable de hacer toda la planificación y gestión de procesos.

*Ambos en Stallings p. 173 - también pp. 454-455

23. Asumiendo el caso de procesadores homogéneos:

(a) ¿Cuál sería el método de planificación más sencillo para asignar CPUs a los procesos?
Compartición de carga: consiste en la existencia de una o varias colas (colas multinivel, por ejemplo) en la que los procesos son asignados a un CPU en la medida en que cada una de estas últimas se libera.

(b) Cite ventajas y desventajas del método escogido

Ventajas:

- La carga se distribuye uniformemente entre los procesadores, asegurando que un procesador no queda ocioso mientras haya trabajo pendiente.
- No se precisa un planificador centralizado; cuando un procesador queda disponible, la rutina de planificación del sistema operativo se ejecuta en dicho procesador para seleccionar el siguiente hilo.
- La cola global puede organizarse y ser accesible usando cualquier esquema de planificación, incluyendo aquellos basados en prioridad o historia de ejecución, o anticipan demandas de procesamiento:

Primero en llegar, primero en ser servido (FCFS). Cuando llega un trabajo, cada uno de sus hilos se disponen consecutivamente al final de la cola compartida. Cuando un procesador pasa a estar ocioso, coge el siguiente hilo listo, que ejecuta hasta que se completa o se bloquea.

Menor número de hilos primero. La cola compartida de listos se organiza como una cola de prioridad, con la mayor prioridad para los hilos de los trabajos con el menor número de hilos no planificados. Los trabajos con igual prioridad se ordenan de acuerdo con qué trabajo llega primero. Al igual que con FCFS, el hilo planificado ejecuta hasta que se completa o se bloquea.

Menor número de hilos primero con expulsión. Se le da mayor prioridad a los trabajos con el menor número de hilos no planificados. Si llega un trabajo con menor número de hilos que un trabajo en ejecución se expulsarán los hilos pertenecientes al trabajo planificado.

Desventajas:

- La cola central ocupa una región de memoria a la que debe accederse de manera que se cumpla la exclusión mutua. De manera que puede convertirse en un cuello de botella si muchos procesadores buscan trabajo al mismo tiempo. Cuando hay sólo un pequeño número de procesadores, es difícil que esto se convierta en un problema apreciable. Sin embargo, cuando el multiprocesador consiste en docenas o quizás cientos de procesadores, la posibilidad de que esto sea un cuello de botella es real.
- Es poco probable que los hilos expulsados retomen su ejecución en el mismo procesador. Si cada procesador está equipado con una caché local, ésta se volverá menos eficaz.
- Si todos los hilos se tratan como un conjunto común de hilos, es poco probable que todos los hilos de un programa ganen acceso a procesadores a la vez. Si se necesita un alto grado de coordinación entre los hilos de un programa, los cambios de proceso necesarios pueden comprometer seriamente el rendimiento.

24. Indique brevemente a que hacen referencia los siguientes conceptos:

(a) Huella de un proceso en un procesador

Durante las diferentes instancias de ejecución de un proceso en un procesador dado, se cargarán datos en diferentes módulos de CPU (caché, TLB, registros, etc.) cuya finalidad es mejorar la performance de ejecución del proceso al posibilitar el acceso rápido a información de uso frecuente (o al menos de posible uso futuro). Si bien buena parte de esta información se pierde en el proceso de cambio de contexto (porque requiere ser sobreescrita por la del proceso entrante), una fracción de estos datos, especialmente aquellos almacenados en caché, permanecerán por un tiempo cercanos al procesador, hasta que pasada la ejecución de múltiples instrucciones de otros procesos, éstos datos finales también sean sobreescritos.

La huella de un proceso, es ese conjunto de datos que corresponden al proceso saliente, y que permanecen por un tiempo en espacios administrados por el CPU, aún cuando el proceso pueda encontrarse bloqueado o finalizado.

(b) Afinidad con un procesador

Es la preferencia de un proceso por ejecutarse en un procesador específico. Se trata de un valor asignado por algunos algoritmos de planificación multiprocesador, en el que al un proceso/hilo entrante,

se le asigna un conjunto de procesadores por los cuales tiene afinidad. Con este indicador, ante la presencia de más de un procesador disponible para atender al proceso se preferirá siempre a aquél que pertenezca al grupo de afinidad del proceso.
Existen dos conceptos específicos asociados a la afinidad:
- Afinidad débil: es la descrita anteriormente, si un proceso está listo y ninguno de los procesadores disponibles pertenece a su grupo de afinidad, igual se asigna el proceso a uno de los CPU libres.
- Afinidad fuerte: en este caso el grupo de afinidad implica exclusividad, es decir que un proceso listo no será ejecutado hasta que no se encuentre disponible un procesador de su grupo de afinidad.

(c) ¿Por qué podría ser mejor en algunos casos que un proceso se ejecute en el mismo procesador?
Dependiendo del caso puede haber varias ventajas, pero en general se trata de mejorar la performance aprovechando precisamente el concepto de huella del proceso. La ejecución consecutiva de ciclos de un proceso en un mismo procesador, aumenta las posibilidades de contar con información requerida (datos, texto, stack, punteros a memoria) ya cargada en el o los niveles de caché con los que este cuenta, reduciendo en gran medida el costo en tiempo que implican la resolución de direcciones y búsqueda en memoria principal.

(d) ¿Puede el usuario en Windows cambiar la afinidad de un proceso? ¿y en GNU/Linux?
Vale aclarar antes que nada que ambos sistemas operativos trabajan con el concepto de afinidad débil por defecto, es decir que a todo nuevo proceso se asigna afinidad con el procesador que está libre para atenderlo, y se intentará que sea el mismo el que lo reciba en sucesivos llamados. La asignación concreta de una afinidad determinada por parte del usuario, tanto en Windows como en GNU/Linux implica el uso de afinidad fuerte.

Windows:
Se puede modificar la afinidad de un proceso con uno o más procesadores desde la pestaña “Detalles” del administrador de tareas.

GNU/Linux:
La afinidad de un proceso se puede asignar con el comando **`taskset`**:
El comando permite asignar afinidad a un proceso en ejecución, o definir la misma al momento de hacer la llamada a otro comando. El concepto clave respecto a los argumentos que acepta este comando es el de máscara.
La máscara es un indicador del conjunto de procesadores con el que se asignará afinidad al proceso, y consiste en una secuencia de bits en los que la posición de los valores en 1 se corresponde con la asignación del procesador con número igual a la posición del bit, siendo el bit menos significativo el correspondiente al CPU 0. De esta forma, por ejemplo, la máscara 001011, asignará afinidad al proceso con los CPU 0,1 y 3. En general, de todas formas, la máscara se expresa en hexadecimal, por lo que el ejemplo anterior sería en realidad 0x0000000A. Cabe señalar que pueden indicarse procesadores que no existan en el sistema, siempre y cuando al menos uno de los asignados realmente esté presente, y se asignarán sólo aquellos procesadores que se encuentren en el sistema. Si ninguno de los procesadores indicados existe, entonces el comando dará un error.

Las posibles formas de utilizar el comando son:
`taskset` *`máscara`* **`comando`** [*`argumentos`*]: ejecutará el **`comando`** indicado con sus respectivos *`argumentos`*, con la afinidad indicada en *`máscara`*.

`taskset -p` *`máscara pid`*: asignará la afinidad indicada en *`máscara`* al proceso con con el *`pid`* indicado.
`taskset -cp` *`procesadores pid`*: asignará la afinidad indicada en *`procesadores`* al proceso con con el *`pid`* indicado. En este caso, la notación de la máscara ya no es una serie de bits, si no que es la enumeración de procesos del grupo de afinidad (siguiendo con el ejemplo dado anteriormente, se indicaría `0,1,3` - la notación también permite rangos: `1-4`, por ejemplo).
`taskset -p` *`pid`*: muestra la afinidad actual del proceso *pid*.

(e) Investigue el concepto de balanceo de carga (load balancing).
El balanceo de carga es una estrategia cuyo foco principal es una distribución aproximadamente equitativa de la carga de trabajo sobre todos los procesadores (aún si son de diferentes capacidades, que la carga sea proporcional a su capacidad en la misma medida para todos). En efecto es la estrategia (de entre varias posibles) que se escoja para asignar un proceso a un procesador determinado.
Según cuál sea el criterio respecto a lo que se considere “carga equitativa”, las características de los procesadores y procesos, así como la información que se posea por anticipado sobre estos últimos (dependencia de determinados recursos, tiempo requerido de CPU, memoria necesaria, etc.), pueden escogerse diversas metodologías para realizar la asignación de procesador a un proceso o conjunto de ellos.

(f) Compare los conceptos de afinidad y balanceo de carga y como uno afecta al otro.
Ambos conceptos están fuertemente relacionados a la historia de ejecución de los procesos entre un conjunto de procesadores, y cómo se ve afectada la performance del sistema con base en las decisiones que se tomen al respecto. Según la implementación, sin embargo, de cada uno, pueden funcionar en detrimento de una u otra forma de concebir la performance. La mejor forma de expresarlo es a través de ejemplos:
- El ejemplo más claro de la contraposición entre ambos conceptos es en el caso de la afinidad fuerte. Ésta asegura que mientras esté en el procesador, cada proceso haga un uso eficiente de su tiempo de cómputo. Sin embargo, dependiendo de la estrategia de balanceo de carga, y la certeza con la que la misma se haya aproximado a la realidad operacional de los procesos al momento de distribuirlos, puede suceder que un o más procesadores agoten sus tareas asignadas mucho antes (los procesos concluyeron mucho más rápido de lo esperado) que otros. Ante una situación de este tipo, los procesadores ahora ociosos no pueden tomar la carga de aquellos que aún se encuentran ocupados, puesto que los procesos, una vez asignada una afinidad, no pueden ser atendidos por procesadores que no pertenezcan a su grupo de afinidad. De esta forma, la carga de trabajo termina siendo no equitativa, aunque garantiza la afinidad a costa de uno o más procesadores ociosos.
- Otro ejemplo posible es cuando se maximiza el balanceo de carga a costa de la afinidad. Esto es, cuando un proceso se encuentra listo, pero no hay procesadores de su grupo de afinidad disponibles, éste es asignado a cualquier otro procesador disponible (el concepto de afinidad débil). De esta forma, la carga sobre cada procesador se mantiene más equitativa, pero una mayor parte del tiempo de atención de los procesos se pierde en la búsqueda de datos que no se encuentran ya tan cerca del CPU, sin contar además que en la medida en que el proceso cambia de procesadores con más frecuencia la huella que pueda dejar sobre cada uno es más pequeña y es más probable que la información sea borrada el el futuro próximo.

25. Si a la tabla del ejercicio 6 la modificamos de la siguiente manera:

JOB	Llegada	CPU	Afinidad

1	0	4	CPU 0
2	2	6	CPU 0
3	3	4	CPU 1
4	6	5	CPU 1
5	8	2	CPU 0

Y considerando que el scheduler de los Sistemas Operativos de la familia Windows utiliza un mecanismo denominado preferred processor (procesador preferido). El scheduler usa el procesador preferido a modo de afinidad cuando el proceso está en estado ready. De esta manera el scheduler asigna este procesador a la tarea si este está libre.

(a) Ejecute el esquema anterior utilizando el algoritmo anterior.
(b) Ejecute el esquema anterior. Pero ahora si el procesador preferido no está libre es asignado a otro procesador. Luego el procesador preferido de cada job es el último en el cual ejecutó.
(c) Para cada uno de los casos calcule el tiempo promedio de retorno y el tiempo promedio de espera.
(d) ¿Cuál de las dos alternativas planteadas es más performante?