Sistemas Operativos Avanzados (MUII)
Ejercicio sobre planificación (14 de abril de 2021)
Planificador CFS

Planteamiento del trabajo

El planificador del procesador (es decir, el módulo del sistema operativo que se encarga de decidir en cada momento qué procesos listos están ejecutándose en los procesadores disponibles) es un componente crítico del sistema ya que de su buen comportamiento van a depender tanto aspectos de rendimiento (los usuarios no instalarán un sistema operativo con un planificador que no extraiga toda la potencia del hardware subyacente) como el grado de satisfacción del usuario (un usuario interactivo no querrá utilizar un sistema operativo si no obtiene una buena experiencia en una reproducción multimedia al sufrir parones por culpa del planificador). En un sistema operativo de propósito general actual el reto del planificador es muy exigente: debe dar un buen servicio en muy distintos tipos de entornos (científicos, empresariales, de uso personal interactivo, de tiempo real no crítico…) para muy diversas plataformas (desde microcontroladores hasta grandes servidores NUMA).

Si revisamos cómo ha sido la evolución de este componente dentro de Linux, la conclusión es un poco decepcionante. En mi opinión, al menos durante mucho tiempo, ha sido el subsistema de Linux más pobre dentro de, por lo demás, un sistema operativo brillante. De hecho, nada menos que hasta la versión 2.6, el algoritmo tenía una complejidad *O(n)* (de ahí el nombre informal de esa versión, que se revisa en la sección 4.17.1 del libro recomendado) siendo *n* el número de procesos existentes en el sistema (aunque parezca sorprendente, ese algoritmo recorría todos los procesos listos cada vez que tenía que seleccionar qué proceso ejecutar y, lo que es peor, al completar todos los procesos listos su rodaja, tenía que reajustar el peso de **todos** los procesos del sistema). Cualquier sistema operativo de hace 50 años incluía un planificador con mucho mejor rendimiento (*O(1)* o, al menos, *O(log(n))*). Me temo que, en este aspecto, y solo en este aspecto, a Linus Torvalds no le pondría muy buena calificación. Quizás detrás de esta anomalía se esconde un cierto menosprecio de Linus hacia ese componente del sistema operativo como expresaba en 2001: "*And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy*".

Curiosamente, esa cita la he extraído del artículo "*The Linux Scheduler: a Decade of Wasted Cores*", que descubre de forma empírica otro fiasco del planificador de Linux: en ciertos escenarios el planificador dejaba procesadores sin usar existiendo procesos listos para ejecutar, lo que es un anatema para un planificador (es como un médico que hubiera roto el juramento hipocrático dejando a un paciente sin atender estando disponible), excepto en entornos de tipo *non-work conserving* (por ejemplo, en un sistema de pago por uso el planificador puede dejar libre un procesador si los usuarios ya han obtenido la cuota de uso abonada), que no era el caso de esos escenarios.

La mayoría de las limitaciones identificadas se han ido subsanando y se puede decir que el algoritmo de planificación actual de Linux, denominado CFS (*Completely Fair Scheduler*), que lleva con nosotros desde 2007, es un algoritmo adecuado que recoge satisfactoriamente todos los requisitos que se plantean actualmente para este tipo de componentes.

Entre esos requisitos, en los últimos tiempos, ha surgido la técnica de la virtualización soportada por el propio sistema operativo, como una solución ligera y ágil para este tipo de entornos. El planificador de un sistema operativo actual debería dar soporte también a este tipo de soluciones basadas en el uso de contenedores.

El trabajo se plantea como dos partes relativamente independientes. En esta primera parte se pretende conocer de forma empírica cuál es el modo de operación del algoritmo CFS. Un algoritmo de planificación de este nivel de sofisticación no tiene siempre un comportamiento que podríamos considerar intuitivo. Mediante

la ejecución de distintos experimentos y el análisis de sus resultados, podemos llegar a entender mejor la idiosincrasia de este algoritmo.

Para afrontar este trabajo, se recomienda consultar las secciones 4.17.3 y 4.18 del libro recomendado.

Planificador CFS de Linux

Antes de presentar este algoritmo, vamos a hacer un muy breve repaso de algunos conceptos relacionados con la planificación del procesador y la evolución de los planificadores.

Tradicionalmente, los planificadores de los sistemas operativos de propósito general han sido una mezcla del uso de prioridades, para permitir establecer grados de urgencia y priorizar los procesos con más entrada/salida e interactivos, con un turno rotatorio (*round robin*), para repartir el tiempo proporcionalmente entre iguales. La mayoría de estos planificadores encajan en el modo de operación de los esquemas de planificación basados en una cola de prioridades multinivel con realimentación, donde los procesos van cambiando de nivel dependiendo de su comportamiento en tiempo de ejecución (procesos con más entrada/salida aumentan su prioridad). La sección 4.9.4 del libro recomendado estudia este tipo de sistemas. Asimismo, en ese documento se revisan algoritmos de sistemas reales que encajan en este esquema: el de Windows (sección 4.13.4), el O(1) de Linux (sección 4.17.2), que fue justo el predecesor de CFS, y el RSDL (sección 4.13.5), propuesto por Con Kolivas, un anestesista australiano que ha diseñado varios algoritmos de planificación para Linux buscando un buen tiempo de respuesta para usuarios interactivos y un carácter autoconfigurable, siendo sus propuestas muy bien acogidas por la comunidad, pero no admitidas de forma oficial.

Se ha producido, sin embargo, un cambio de paradigma en el diseño de planificadores a través del concepto de planificación con cuotas equitativas (*fair-share scheduling*), introducido en la sección 4.3.4, y que, a continuación, se presenta de forma muy breve, pues es un aspecto clave en este ejercicio al estar imbuido en la esencia de CFS.

Los planificadores tradicionales reparten el tiempo equitativamente entre los procesos existentes. ¿Es eso justo si un usuario tiene un único proceso y el otro posee 99? ¿Hubiera sido mejor que el primer proceso se llevará el 50% del uso del procesador en vez del 1%?

Ninguna opción es mejor que la otra: dependerá de cada caso. Lo que se busca, por tanto, es un algoritmo de planificación que permita configurarlo; pero no solo en ese caso de competencia relativamente trivial sino en escenarios jerárquicos más sofisticados, tal como, por ejemplo, un sistema donde se pretende que los procesos de los alumnos se lleven el 70% del uso del procesador, repartido a su vez a la mitad entre los alumnos de grado y los de posgrado.

Este reparto jerárquico configurable del procesador entre grupos de procesos se extiende a todos los demás recursos del sistema, tal como se analiza en la sección 4.15.

Una aplicación directa importante de este esquema es el soporte de la planificación de contenedores, ya que permite establecer cuotas de reparto de recursos entre los contenedores y, a su vez, el reparto de esa cuota entre los procesos de cada contenedor.

El algoritmo CFS, que se estudia en la sección 4.17.3, es el primer algoritmo con cuotas equitativas incluido en Linux y se basa en repartir equitativamente el procesador entre los procesos existentes teniendo en cuenta las posibles relaciones jerárquicas configuradas para la asignación de cuotas del procesador. En aras de la equidad, CFS asigna el mismo tiempo de uso de procesador a todos los procesos, pero gestionando un tiempo virtual que se adapta al peso relativo de cada proceso (cuando ejecuta un proceso con mayor peso el tiempo corre más lento, por lo que en el mismo tiempo virtual puede hacer más cosas). La sección 4.9.6 presenta otros algoritmos basados en el concepto de tiempo virtual.

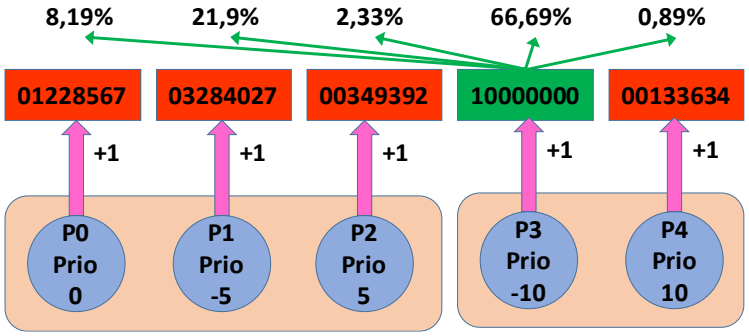
Ejercicio

Para medir cómo se reparten el procesador varios procesos durante un experimento se va a seguir una estrategia, descrita en la sección 4.18.1, que, aunque no sea de todo precisa, es sencilla y no requiere realizar ningún tipo de instrumentación sofisticada y compleja en los programas de prueba.

La idea es que todos los procesos del experimento van a incrementar el valor de una variable propia que actúa de contador intentando alcanzar un cierto valor máximo, pero de forma que el primero que lo alcance imprimirá la cuota de procesador que ha obtenido cada proceso a partir del valor que ha alcanzado el contador del mismo.

Hay que resaltar que se trata de una estimación poco precisa, que calcula el porcentaje de uso del procesador por parte de cada proceso del experimento suponiendo que el procesador está dedicado únicamente a ejecutar el experimento. En cualquier caso, consideramos que es un enfoque adecuado para ilustrar de una manera sencilla los aspectos que se quieren reforzar en este ejercicio.

La siguiente figura ilustra el modo de operación de esta estrategia mostrando un experimento con dos aplicaciones ejecutando tres y dos procesos, respectivamente. En la misma se puede apreciar que cuando el proceso *P3* completa su labor (su contador llega al valor máximo especificado en ese experimento: 10000000) realiza una estimación sobre qué cuota de procesador ha obtenido cada proceso e imprime el resultado.



La siguiente fórmula muestra cómo se llevaría a cabo ese cálculo para el primer proceso del experimento:

$$cuota\ de\ P0 = 100 \times \frac{1228567}{1228567 + 3284027 + 349392 + 10000000 + 133634} = 8{,}19\%$$

Como material de apoyo se proporciona un programa (*programa1*) que usa el servicio `nice`, presentado en la sección 4.18.1, para cambiar la prioridad de un proceso e implementa la lógica correspondiente a la estrategia que se acaba de describir en la que se basan los experimentos de este ejercicio.

En el programa, tanto el proceso padre como los hijos creados ejecutan un bucle que incrementa el valor de un contador hasta que uno de ellos alcance el valor máximo estipulado. El programa está diseñado teniendo en mente que puede haber varias instancias del mismo activas durante un experimento. En el programa se hace uso de la técnica de proyección de ficheros para hacer que los contadores de iteraciones de los procesos de las distintas instancias puedan ser visibles por todos los procesos, de manera que, cuando termina el proceso *ganador*, va a poder calcular e imprimir la estimación del uso del procesador de los procesos de todas las instancias a partir del valor de sus contadores. El programa, que no va a recibir el premio a la mejor interfaz, recibe los siguientes argumentos:

- `num_iter`: indica cuál es el valor máximo que tiene que alcanzar el contador. Cuanto mayor sea más precisa será la estimación. **Si los valores obtenidos durante un experimento no son razonables, pruebe a aumentar significativamente el número de iteraciones**. En cualquier caso, en cada ejecución de un mismo experimento, se obtendrán diferentes valores, dada la estrategia usada.
- `n_proc_total`: informa de cuántos procesos tomarán parte en el experimento, incluyendo los de todas las instancias.
- `n_proc_prog`: especifica cuántos procesos habrá en esta instancia, incluyendo el proceso padre inicial.
- `id_primero`: indica cuál es el identificador asociado al primer proceso de esta instancia. Los procesos en un experimento se van a identificar con valores sucesivos a partir del 0. El proceso 0 corresponderá al primer proceso de la primera instancia. Si esa primera instancia tiene 3 procesos, el identificador del primer proceso de la segunda instancia será 3 y así sucesivamente.
- `prio_proc`: especifica cuáles son las prioridades de los procesos de esta instancia. Habrá un valor para cada proceso, correspondiendo el primero al proceso padre inicial.

Recuerde que para que el programa pueda aumentar la prioridad de un proceso o bien se ejecuta en modo superusuario o, si se prefiere que lo haga un usuario normal, se necesitará utilizar alguna de las técnicas explicadas en la sección 4.18.1. Por ejemplo, se puede añadir a ese programa la capacidad de aumentar la prioridad de un proceso con independencia de qué usuario lo ejecute (las capacidades las estudiaremos en el capítulo dedicado a la protección):

```
sudo setcap cap_sys_nice+eip programa1
```

A continuación, se muestra la secuencia de mandatos correspondiente al experimento que se mostró en la figura previa (para ejecutarla de un tirón, puede realizar una operación de copia y pega de la misma, especificarla en una única línea de mandatos o, quizás lo más cómodo, incluirla en un *script*).

```
# repaso de args: 5 procesos en total; 3 en esta instancia; el primero con ID 0;
# el primero con prioridad 0; el 2º prioridad -5; el 3º prioridad 5.
taskset 1 ./programa1 100000000 5 3 0 0 -5 5 2>/dev/null &

# repaso de args: 5 procesos en total; 2 en esta instancia; el primero con ID 3;
# el primero con prioridad -10; el 2º prioridad 10.
taskset 1 ./programa1 100000000 5 2 3 -10 10 2>/dev/null &

wait # espera que finalicen las dos aplicaciones
```

Nótese que se ha usado el mandato `taskset`, que es específico de Linux (el resto es aplicable a cualquier sistema de la familia UNIX) y fija la máscara de afinidad estricta del proceso, tal como se presenta en la sección 4.18.3, para asegurarnos de que todos los procesos ejecutan en el mismo procesador (máscara igual a 1, que especifica que solo se ejecuta en el primer procesador), lo que permite analizar mejor el reparto de procesador que realiza el esquema de planificación dependiendo de la prioridad. Observe que para mejorar la estimación se han usado 100000000 iteraciones en vez de 10000000, como ocurría en la figura.

El ejercicio plantea ejecutar distintos experimentos y analizar de forma razonada los resultados, que se deben incluir también como parte de la respuesta:

1. Como punto de control inicial, ejecute el experimento anterior.
2. Repítalo, pero haciendo que todos los procesos tengan la misma prioridad. ¿Influye el valor de la prioridad en el resultado? ¿Es lo mismo que todos tengan la máxima prioridad o que posean la mínima?

3. Realice dos experimentos con procesos que tengan prioridades consecutivas, pero en cada experimento en una zona diferente del espectro (recuerde que las prioridades se extienden desde la máxima -20 a la mínima 19). ¿Influye la ubicación en el espectro en el reparto del procesador?
4. Como se explica en la sección 4.17.3, que presenta el algoritmo CFS, este esquema usa, por defecto, la técnica del *autogroup*, que crea un grupo implícito para los procesos de una sesión (*login*) de un usuario. Para emular sesiones independientes, ejecute cada instancia del segundo experimento con el mandato *setsid* (con la opción –*w* para que espere).
5. Para ver todavía más claro el efecto de *autogroup*, realice un experimento con dos instancias con un único proceso cada una: la primera con un proceso de prioridad máxima y la segunda con uno de mínima. ¿Cree que estos resultados pueden ser sorprendentes para los usuarios?
6. Repita el ejemplo previo desactivando *autogroup* (*/proc/sys/kernel/sched_autogroup_enabled*).
7. Vuelva a repetir el experimento 4 pero haciendo que el *autogroup* de la primera sesión (la primera instancia) tenga una prioridad de -10. Nótese que, por defecto, un *autogroup* tiene una prioridad 0. Este valor puede cambiarse escribiendo la prioridad en el fichero */proc/PID/autogroup*, siendo PID el identificador de cualquier proceso del *autogroup*. Para hacer la prueba, puede crearse una nueva versión del programa que haga esa escritura al principio del mismo y usarla para activar la primera instancia:

```
char cmd[256];
sprintf(cmd, "echo -10 > /proc/%d/autogroup", getpid());
system(cmd);
```

Como se mencionó previamente y se estudia en la sección 4.17.3, Linux proporciona un esquema de planificación con cuotas equitativas que, además de crear grupos implícitos para representar las distintas sesiones (*autogroup*), permite que un usuario con los permisos pertinentes pueda crear de forma explícita sus propios grupos de procesos para administrar a su conveniencia el reparto del procesador entre los mismos, usando para ello el mecanismo denominado *cgroups* presentado en la sección 4.17.5.
En esa sección se explica el modo de operación de este mecanismo, donde cada grupo se representa por un directorio dentro de la jerarquía del subsistema (recurso) de interés (en este caso, *cpu*) tal que ese directorio contiene ficheros que permiten realizar la configuración de los parámetros relevantes, así como añadir procesos al grupo escribiendo su PID en el fichero `cgroup.procs`.

Así, por ejemplo, se crearía un grupo denominado *grupo1*:

```
mkdir /sys/fs/cgroup/cpu/grupo1 # crea un grupo
```

Para facilitar la incorporación a un grupo del proceso asociado a la instancia se ha optado por que el programa escriba por la salida de error su PID, lo que permite la redirección directa:

```
taskset 1 ./programa1 1000 5 3 0 0 0 0 2>/sys/fs/cgroup/cpu/grupo1/cgroup.procs &
```

En este ejercicio, estamos interesados en el subsistema `cpu`, que hace posible configurar el reparto del procesador entre los grupos hermanos mediante el atributo `cpu.shares` que especifica el peso de cada grupo, siendo 1024 el valor por defecto.

8. Repita el primer experimento, pero creando un grupo por cada instancia, no modificando `cpu.shares`.
9. Vuelva a repetir el experimento pero asignando `cpu.shares` de 2048 a la primera instancia y 512 a la segunda. ¿Qué porcentaje aproximado se lleva cada grupo?
10. Para ver el efecto de mezclar grupos implícitos (*autogroup*) y explícitos, repita el cuarto experimento, pero incluyendo todos los procesos en un mismo grupo explícito.

Plazo y modo de entrega

El plazo de entrega del trabajo es el 8 de junio de 2021.

La entrega se realiza en *triqui* ejecutando el mandato:

entrega.soa planificacion.2021

Este mandato realizará la recolección del fichero *~/DATSI/SOA/planificacion.2021/memoria.pdf*, que debe incluir la solución del ejercicio.