7

Planificación: Introducción

A estas alturas, los **mecanismos** de bajo nivel de los procesos en ejecución (como por ejemplo, el cambio de contexto) ya deberían estar claros; de no ser así, retrocedé uno o dos capítulos y volvé a leer la descripción de cómo funcionan estas cosas. Aún así, todavía nos falta comprender las **políticas** de alto nivel que emplea el planificador de un SO. Eso es precisamente lo que vamos a hacer ahora, presentando una serie de **políticas de planificación** (a veces llamadas **disciplinas**) que varias personas inteligentes y trabajadoras fueron desarrollando a lo largo de los años.

Los orígenes de la planificación, de hecho, preceden a los sistemas informáticos; los primeros enfoques fueron tomados del campo de gestión de operaciones y aplicados a las computadoras. Esta realidad no debería ser una sorpresa: las líneas de montaje y muchos otros esfuerzos humanos también requieren planificación y comparten muchas de las mismas preocupaciones, incluyendo un deseo por eficiencia similar a la de un láser. Y así, nuestro problema:

LA CUESTIÓN: CÓMO DESARROLLAR UNA POLÍTICA DE PLANIFICACIÓN

¿Cómo deberíamos desarrollar un marco básico para pensar en las políticas de planificación? ¿Cuáles son las suposiciones clave? ¿Qué métricas son importantes? ¿Qué enfoques básicos se utilizaron en los primeros sistemas informáticos?

7.1 Suposiciones Sobre la Carga de Trabajo

Antes de entrar en la gama de políticas posibles, primero plantearemos algunas suposiciones simplificadoras sobre los procesos que se ejecutan en el sistema, a veces denominados colectivamente como

la **carga de trabajo**. Determinar la carga de trabajo es una parte fundamental de la creación de políticas, y cuanto más se sepa sobre la carga de trabajo, más afinada podrá ser la política.

Las suposiciones de la carga de trabajo que planteamos son en su mayoría poco realistas, pero eso está bien (por ahora), pues a medida que avancemos las iremos flexibilizando y, con el tiempo, desarrollaremos lo que conoceremos como... (pausa dramática)... una **disciplina de planificación completamente funcional**[1].

Haremos las siguientes suposiciones sobre los procesos, a veces llamados **trabajos**, que se ejecutan en el sistema:

1. Cada trabajo se ejecuta durante la misma cantidad de tiempo.
2. Todos los trabajos llegan al mismo tiempo.
3. Una vez iniciado, cada trabajo se ejecuta hasta su finalización.
4. Todos los trabajos usan solo la CPU (es decir, no realizan E/S)
5. Se conoce el tiempo de ejecución de cada trabajo.

Hemos dicho que muchas de estas suposiciones eran poco realistas, pero así como algunos animales son más parecidos que otros en Rebelión en la granja de Orwell [O45], algunas suposiciones son menos realistas que otras en este capítulo. En particular, puede que te moleste el hecho de que se conozca el tiempo de ejecución de cada trabajo: esto haría que el planificador sea omnisciente, lo que, aunque sería genial (probablemente), no tiene muchas posibilidades de suceder pronto.

7.2 Métricas de Planificación

Además de hacer suposiciones sobre la carga de trabajo, también nos hace falta algo que nos permita comparar diferentes políticas de planificación: una **métrica de planificación**. Una métrica es algo que usamos para medir algo, y hay varias métricas diferentes que tienen sentido en el mundo de la planificación.

Por ahora, sin embargo, hagamos nuestra vida más sencilla teniendo simplemente una única métrica: el **tiempo de entrega**. El tiempo de entrega de un trabajo se define como el momento en que se completa el trabajo menos el momento en que el trabajo llegó al sistema. Más formalmente, el tiempo de entrega $T_{entrega}$ es:

$$T_{entrega} = T_{finalizacion} - T_{llegada}$$

[1] Dicho de la misma forma que "Una Estrella de la Muerte completamente funcional"

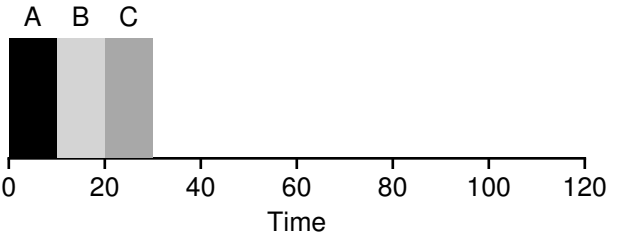

Figure 7.1: **Ejemplo Simple de FIFO**

Como asumimos que todos los trabajos llegan al mismo tiempo, entonces por el momento diremos que $T_{llegada} = 0$ y por lo tanto $T_{entrega} = T_{finalizacion}$. Esto irá cambiando a medida que relajemos las suposiciones antes mencionadas.

Notar que el tiempo de entrega es una métrica de **rendimiento**, que será nuestro enfoque principal en este capítulo. Otra métrica de interés puede ser la **justicia**, medida (por ejemplo) por el **Índice de Justicia de Jain** [J91]. El rendimiento y la justicia suelen estar en desacuerdo en la planificación; un planificador, por ejemplo, puede optimizar el rendimiento, pero a costa de evitar que se ejecuten algunos trabajos, reduciendo así la justicia. Este problema nos enseña que la vida no siempre es perfecta.

7.3 Primero En Entrar, Primero En Salir (FIFO)

El algoritmo más básico que podemos implementar es conocido como la política **Primero en Entrar, Primero en Salir** (**FIFO**, en inglés First In, First Out) o a veces, como **Primero en Llegar, Primero en Ser Atendido** (**FCFS**, First Come, First Served). FIFO tiene varias propiedades positivas: es claramente simple y, por lo tanto, fácil de implementar. Y dadas nuestras suposiciones, funciona bastante bien.

Hagamos un ejemplo rápido juntos. Imaginemos que llegan tres trabajos al sistema, A, B y C, aproximadamente al mismo tiempo ($T_{llegada} = 0$). Como FIFO tiene que poner algún trabajo primero, supongamos que si bien todos llegaron simultáneamente, A llegó justo un momento antes que B, que a su vez llegó un momento antes que C. Supongamos también que cada trabajo se ejecuta durante 10 segundos. ¿Cuál será el **tiempo medio de entrega** de estos trabajos?

En la figura 7.1, se puede ver que A terminó en 10, B en 20 y C en 30. Por lo tanto, el tiempo medio de entrega para los tres trabajos es simplemente $\frac{10+20+30}{3} = 20$. Calcular el tiempo de entrega es así de fácil.

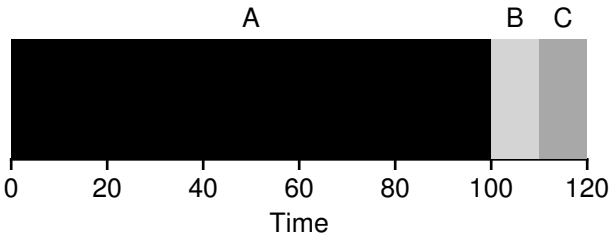

Figure 7.2: **Por Qué FIFO No Es Tan Genial**

Ahora flexibilicemos una de nuestras suposiciones. En particular, olvidemos el supuesto 1 y, por lo tanto, dejemos de suponer que todos los trabajos se ejecutan durante la misma cantidad de tiempo. ¿Cómo funciona FIFO ahora? ¿Qué tipo de carga de trabajo podrías construir para que FIFO funcione mal? (*pensalo bien antes de seguir leyendo... seguí pensando... ¡¿ya está?!*)

Ya lo deberías haber descifrado para este momento, pero por si acaso, hagamos un ejemplo para mostrar cómo los trabajos de diferentes longitudes pueden generar problemas en la planificación FIFO. En particular, supongamos de nuevo tres trabajos (A, B y C), pero esta vez A se ejecuta durante 100 segundos mientras que B y C se ejecutan durante 10 cada uno.

Como se puede ver en la Figura 7.2, el trabajo A corre primero durante 100 segundos completos antes de que B o C tengan la oportunidad de ejecutarse. Por lo tanto, el tiempo medio de entrega del sistema es alto: unos dolorosos 110 segundos ($\frac{100+110+120}{3} = 110$).

Este problema se conoce generalmente como el **efecto de convoy** [B+79], en el que una serie de consumidores potenciales de algún recurso, relativamente cortos, se ponen en cola detrás de un consumidor de gran peso. Puede que este escenario de planificación te suene a cuando hay una única fila en un supermercado y lo que sentís cuando ves a la persona en frente tuyo con tres carritos llenos de provisiones, y una chequera en mano; va a demorar un rato[2].

Entonces, ¿qué deberíamos hacer? ¿Cómo podemos desarrollar un mejor algoritmo para hacerle frente a nuestra nueva realidad de trabajos que se ejecutan durante diferentes períodos de tiempo? Pensalo un rato y después seguí leyendo.

[2]La acción recomendada en estos casos es cambiar rápidamente de fila o tomar una respiración larga, profunda y relajante. Así es, inhalá, exhalá. Todo va a estar bien, no te preocupes.

TIP: EL PRINCIPIO DE SJF
La idea de SJF representa un principio de planificación general que se puede aplicar a cualquier sistema en el que el tiempo de entrega percibido por cliente (o, en nuestro caso, trabajo) sea importante. Pensá en cualquier fila en la que hayas esperado: si el establecimiento en cuestión se preocupa por la satisfacción del cliente, es probable que haya tenido en cuenta SJF. Por ejemplo, los supermercados suelen tener una línea de "diez artículos o menos" para garantizar que los compradores que solo tienen unas pocas cosas para comprar no se queden atrapados detrás de la familia que se prepara para el próximo invierno nuclear.

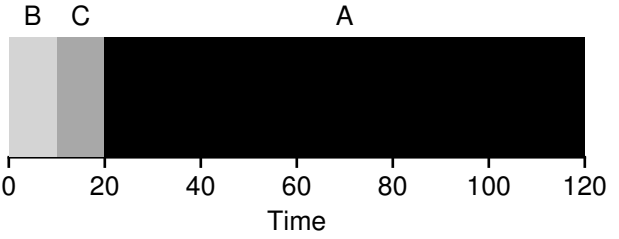

Figure 7.3: **Ejemplo Simple de SJF**

7.4 Trabajo Más Corto Primero (SJF)

Resulta que hay un enfoque muy simple que resuelve este problema; de hecho, la idea fue robada de la investigación de operaciones [C54,PV56] y fue aplicada a la planificación de trabajos en sistemas informáticos. Esta nueva disciplina de planificación se conoce como la del **Trabajo Más Corto Primero** (**SJF**, del inglés Shortest Job First), y su nombre debe ser fácil de recordar ya que describe a la política de manera bastante completa: primero ejecuta el trabajo más corto, luego el siguiente, y así sucesivamente.

Tomemos nuestro ejemplo anterior, pero con SJF como nuestra política de planificación. La Figura 7.3 muestra los resultados de ejecutar A, B y C. El diagrama debería aclarar por qué SJF tiene un rendimiento mucho mejor con respecto al tiempo medio de entrega. Simplemente ejecutando B y C antes que A, SJF reduce el tiempo medio de entrega de 110 segundos a 50 ($\frac{(10+20+120}{3} = 50$), una mejora de más de un factor de dos.

De hecho, dada nuestra suposición de que todos los trabajos llegan al mismo tiempo, podríamos demostrar que SJF es un algoritmo

APARTE: PLANIFICADORES APROPIATIVOS
En los viejos tiempos de la computación por lotes, se desarrollaron varios planificadores **no apropiativos**; dichos sistemas corrían cada trabajo hasta su finalización antes de considerar si ejecutar un trabajo nuevo. Prácticamente todos los planificadores modernos son **apropiativos** y están dispuestos a detener la ejecución de un proceso para ejecutar otro. Esto implica que el planificador emplea los mecanismos que conocimos anteriormente; en particular, el planificador puede realizar un **cambio de contexto**, deteniendo temporalmente un proceso en ejecución y reanudando (o iniciando) otro.

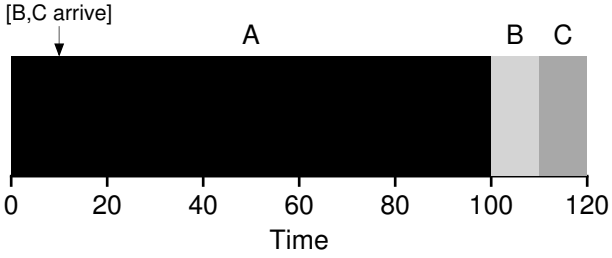

Figure 7.4: **SJF Con Llegadas Tardías de B y C**

de planificación **óptimo**. Sin embargo, está es una clase de sistemas, no de teoría o de investigación de operaciones; las demostraciones no están permitidas.

Así, con SJF llegamos a un buen enfoque para la planificación, pero nuestras suposiciones siguen siendo poco realistas. Relajemos otra más. En particular, podemos elegir el supuesto 2 y ahora asumir que los trabajos pueden llegar en cualquier momento en lugar de todos a la vez. ¿A qué problemas conduce esto?

(*Otra pausa para pensar... ¿estás pensando? Vamos, vos podés*)

Nuevamente podemos ilustrar el problema con un ejemplo. Esta vez, supongamos que A llega en $t = 0$ y necesita ejecutarse durante 100 segundos, mientras que B y C llegan en $t = 10$ y cada uno necesita ejecutarse durante 10 segundos. Con SJF puro, obtendremos la planificación que se observa en la Figura 7.4.

Como se ve en la figura, aunque B y C llegaron poco después de A, todavía se ven obligados a esperar hasta que A se haya completado y, por lo tanto, sufren del mismo problema de convoy. El tiempo medio de entrega para estos tres trabajos es de 103,33 segundos $\frac{100+(110-10)+(120-10)}{3}$. ¿Qué puede hacer un planificador al respecto?

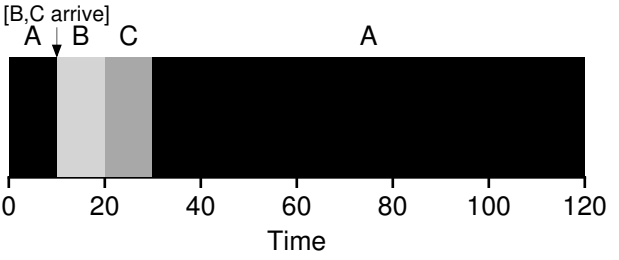

Figure 7.5: **Ejemplo simple de STCF**

7.5 Trabajo de Menor Tiempo Restante Primero (STCF)

Para abordar este aspecto, debemos soltar el supuesto 3 (que los trabajos deben ejecutarse hasta su finalización), así que hagámoslo. También necesitamos algo de maquinaria dentro del propio planificador. Como habrás adivinado, dada nuestra discusión anterior sobre interrupciones de temporizador y cambios de contexto, ciertamente hay algo más que el planificador puede hacer cuando llegan B y C: puede **detener** el trabajo A y decidir ejecutar otro trabajo en su lugar, quizás continuando A más tarde. SJF, según nuestra definición, es un planificador **no apropiativo** y, por lo tanto, sufre los problemas descritos anteriormente.

Afortunadamente, hay un planificador que hace exactamente eso: agrega apropiación a SJF, conocido como el planificador del **Trabajo de Menor Tiempo Restante Primero** (**STCF**, del inglés Shortest Time-to-Completion First) o del **Trabajo Más Corto Primero con Apropiación**) (**PSJF**, Preemptive Shortest Job First) [CK68]. Cada vez que un nuevo trabajo ingresa al sistema, el planificador STCF determina a cuál de los trabajos restates (incluyendo el nuevo trabajo) le queda el menor tiempo hasta finalizar, y lo elige para ser ejecutado. Por lo tanto, en nuestro ejemplo, STCF habría detenido a A y ejecutado a B y a C hasta su finalización; y recién cuando hayan terminado, habría elegido ejecutar lo que quede de A. La Figura 7.5 muestra un ejemplo de esto.

El resultado es un tiempo medio de entrega mucho mejor: 50 segundos ($\frac{(120-0)+(20-10)+(30-10)}{3}$)). Y como antes, dadas nuestras nuevas suposiciones, STCF es demostrablemente óptimo; dado que SJF es óptimo si todos los trabajos llegan al mismo tiempo, probablemente puedas ver la intuición detrás de la optimalidad de STCF.

7.6 Una Nueva Métrica: El Tiempo de Respuesta

Así, si supiéramos la duración de cada trabajo, y si los traba-

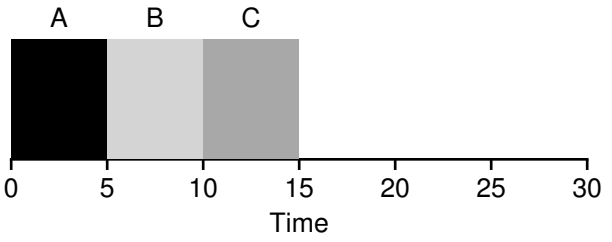

Figure 7.6: **SJF de Nuevo (Malo Para el Tiempo de Respuesta)**

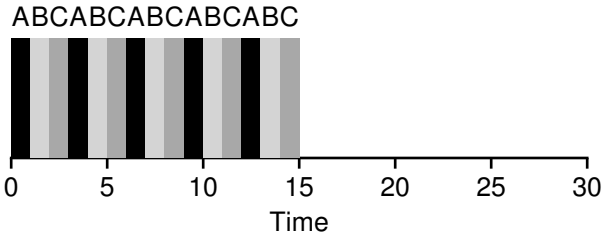

Figure 7.7: **Round Robin (Bueno Para el Tiempo de Respuesta)**

jos usaran solamente la CPU, y además nuestra única métrica fuera el tiempo de entrega, STCF sería una excelente política de planificación. De hecho, para varios de los primeros sistemas informáticos por lotes, estos tipos de algoritmos de planificación tenían cierto sentido. Pero todo esto cambió cuando se introdujeron las máquinas de tiempo compartido. Ahora, los usuarios se sentarían en una terminal y le exigirían al sistema un rendimiento interactivo. Entonces nació una nueva métrica: el **tiempo de respuesta**.

Definimos el tiempo de respuesta como el tiempo desde que el trabajo llega a un sistema hasta la primera vez que es elegido para ser ejecutado[3]. Más formalmente:

$$T_{respuesta} = T_{1ra-ejecucion} - T_{llegada}$$

Por ejemplo, si tuviésemos el programa de la Figura 7.5 (con A llegando en el tiempo 0, y B y C en el tiempo 10), el tiempo de respuesta de cada trabajo sería el siguiente: 0 para el trabajo A, 0 para B y 10 para C (media: 3,33).

[3]Algunos lo definen de manera ligeramente diferente, por ejemplo, incluyendo también el tiempo hasta que el trabajo produce algún tipo de "respuesta"; nuestra definición es la versión del mejor de los casos, esencialmente asumiendo que el trabajo produce una respuesta instantánea.

Como estarás pensando, STCF y las disciplinas relacionadas no son particularmente buenas con el tiempo de respuesta. Si tres trabajos llegan al mismo tiempo, por ejemplo, el tercer trabajo tiene que esperar a que los dos trabajos anteriores se ejecuten en su totalidad antes de ser elegido por primera vez. Si bien es excelente para el tiempo de entrega, este acercamiento es bastante malo para el tiempo de respuesta y para la interactividad. Es más, imaginate sentado en una terminal, escribiendo, y teniendo que esperar 10 segundos para ver una respuesta del sistema simplemente porque se eligió ejecutar otro trabajo antes que el tuyo: no sería demasiado agradable.

Por lo tanto, nos quedamos con otro problema: ¿Cómo podemos construir un planificador que sea sensible al tiempo de respuesta?

7.7 Round Robin

Para resolver este problema, presentamos un nuevo algoritmo de planificación, denominado clásicamente como planificación **Round-Robin** (**RR**) [K64]. La idea básica es simple: en lugar de ejecutar trabajos hasta su finalización, RR ejecuta cada trabajo durante un **segmento de tiempo** (a veces llamado **quantum de planificación**) y luego cambia al siguiente trabajo en la cola de ejecución. Esto lo hace repetidamente hasta que se terminan todos los trabajos. Por esta razón, RR se denomina a veces como **división de tiempo**. Notemos que la duración de un segmento de tiempo debe ser un múltiplo del período de interrupción del temporizador; por lo tanto, si el temporizador se interrumpe cada 10 milisegundos, el segmento de tiempo podría ser de 10, 20 o cualquier otro múltiplo de 10 ms.

Para comprender RR con más detalle, veamos un ejemplo. Supongamos que tres trabajos A, B y C llegan al mismo tiempo al sistema y que cada uno desea ejecutarse durante 5 segundos. Un planificador SJF ejecuta cada trabajo hasta su finalización antes de ejecutar otro (Figura 7.6). Por el contrario, RR con un segmento de tiempo de 1 segundo recorrería los trabajos rápidamente (Figura 7.7).

El tiempo promedio de respuesta de RR es de $\frac{0+1+2}{3} = 1$; para SJF, el tiempo medio de respuesta es de $\frac{0+5+10}{3} = 5$.

Como verás, la duración del segmento de tiempo es fundamental en RR. Cuanto más corto sea, mejor será el rendimiento de RR según la métrica del tiempo de respuesta. Sin embargo, si el segmento de tiempo es demasiado corto puede resultar problemático: de repente, el costo del cambio de contexto dominaría el rendimiento general. Por lo tanto, decidir la duración del segmento de tiempo presenta un intercambio que el diseñador del sistema debe estar dispuesto a hacer; tiene que ser suficientemente largo como para **amortizar** el costo del cambio, pero no tan largo como para que el sistema ya no responda.

TIP: LA AMORTIZACIÓN PUEDE REDUCIR COSTOS
La técnica general de amortización se usa comúnmente en sistemas en los que existe un costo fijo para alguna operación. Al incurrir en ese costo con menos frecuencia (es decir, al realizar la operación menos veces), se reduce el costo total del sistema. Por ejemplo, si se asigna el segmento de tiempo a 10 ms y el costo del cambio de contexto es de 1 ms, aproximadamente el 10% del tiempo se gasta en el cambio de contexto y, por lo tanto, se pierde. Si queremos amortizar este costo, podemos aumentar el segmento de tiempo, por ejemplo, a 100 ms. En este caso, se gasta menos del 1% del tiempo en el cambio de contexto y, por lo tanto, se ha amortizado el costo de la división del tiempo.

Notá que el costo del cambio de contexto no surge únicamente de las acciones del SO de guardar y restaurar algunos registros. Cuando los programas se ejecutan, acumulan una gran cantidad de estado en cachés de CPU, TLB, predictores de saltos y otro hardware en chip. Cambiar a otro trabajo hace que este estado se vacíe y se introduzca un nuevo estado relevante para el trabajo que se esté ejecutando en el momento, lo que puede exigir un notable costo de rendimiento [MB91].

RR, con un segmento de tiempo razonable, es, por tanto, un planificador excelente si el tiempo de respuesta es nuestra única métrica. Pero, ¿qué pasa con nuestro viejo amigo, el tiempo de entrega? Veamos de nuevo nuestro ejemplo anterior. A, B y C, cada uno con tiempos de ejecución de 5 segundos, llegan al mismo tiempo, y el planificador es de tipo RR con un (largo) segmento de tiempo de 1 segundo. Podemos ver en la imagen de arriba que A termina en 13, B en 14 y C en 15, para una media de 14. ¡Feísimo!

No es de extrañar, entonces, que RR sea de hecho una de las peores políticas si nuestra métrica es el tiempo de entrega. Intuitivamente, esto tiene sentido: lo que hace RR es estirar cada trabajo tanto como pueda, ejecutando cada trabajo por muy poco tiempo antes de pasar al siguiente. Debido a que el tiempo de entrega solo se preocupa por cuándo terminan los trabajos, RR es casi pesimista, incluso peor que el simple FIFO en muchos casos.

Generalmente, cualquier política (como RR) que sea **justa**, es decir, que divida uniformemente la CPU entre los procesos activos en una escala de tiempo pequeña, tendrá un desempeño deficiente en métricas como el tiempo de entrega. De hecho, este trato es inherente: si estás dispuesto a ser injusto, podés ejecutar los trabajos más cortos hasta su finalización, pero a costa del tiempo de respuesta; si, en cambio, valorás la justicia, el tiempo de respuesta se reduce, pero a costa del tiempo de entrega. Este tipo de intercambio es común en

los sistemas; no se puede estar en la misa y en la procesión a la vez [4].

TIP: SUPERPONER PERMITE UNA MAYOR UTILIZACIÓN
Cuando sea posible, **superponé** las operaciones para maximizar la utilización de los sistemas. La superposición es útil en varios dominios diferentes, incluyendo la realización de E/S en disco o el envío de mensajes a máquinas remotas; en cualquier caso, comenzar la operación y cambiar a otro trabajo es una buena idea, y aumenta la utilización y la eficiencia general del sistema.

Hemos desarrollado dos tipos de planificadores. El primer tipo (SJF, STCF) optimiza el tiempo de entrega, pero es malo para el tiempo de respuesta. El segundo tipo (RR) optimiza el tiempo de respuesta, pero es malo para el tiempo de entrega. Y todavía tenemos dos suposiciones que debemos aflojar: el supuesto 4 (que los trabajos no realizan E/S) y el supuesto 5 (que se conoce el tiempo de ejecución de cada trabajo). A continuación, abordaremos estas suposiciones.

7.8 Incorporando E/S

Primero relajemos el supuesto 4: obviamente todos los programas realizan E/S. Imaginate un programa que no tome ninguna entrada: siempre produciría la misma salida. Imaginate ahora uno sin salida: es como el árbol que cae en el bosque, sin que haya nadie para verlo; no importa que se haya ejecutado.

Claramente el planificador tiene que tomar una decisión cuando algún trabajo inicia una solicitud de E/S, pues el trabajo que se está ejecutando deja de usar la CPU durante la E/S, y se queda **bloqueado** esperando su finalización. Si la E/S se envía a una unidad de disco duro, es posible que el proceso se bloquee durante unos milisegundos o más, dependiendo de la carga de E/S de la unidad en ese momento. Por lo tanto, quizás sea mejor que el planificador aproveche este tiempo para ejecutar otro trabajo en la CPU.

El planificador también debe tomar una decisión cuando se termina la E/S. Cuando esto ocurre, se genera una interrupción y se ejecuta el SO, moviendo el proceso que emitió la E/S del estado bloqueado al estado listo. Por supuesto, podría incluso decidir seguir ejecutando el mismo trabajo en ese momento. ¿Cómo debería tratar el SO a cada trabajo?

Para comprender mejor este problema, supongamos que tenemos dos trabajos, A y B, que necesitan cada uno 50 ms de tiempo de CPU.

[4] Sorprendentemente, hay una página de wikipedia sobre este dicho; y lo que es aún más sorprendente, es que es una lectura divertida [W15]. Como dicen en italiano, no se puede *Avere la botte piena e la moglie ubriaca*.

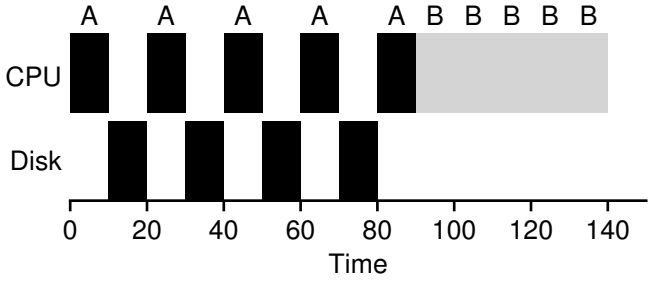

Figure 7.8: **Mal Uso de los Recursos**

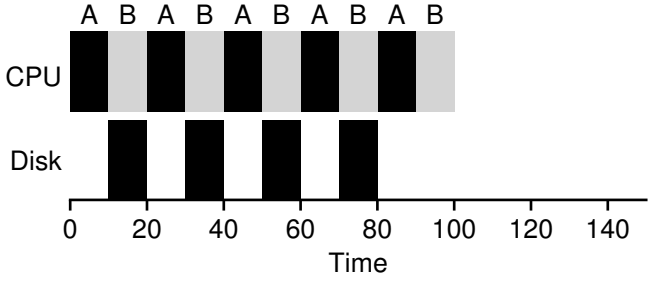

Figure 7.9: **Superponer Permite un Mejor Uso de los Recursos**

Sin embargo, hay una diferencia obvia: A se ejecuta durante 10 ms y después emite una solicitud de E/S (supongamos que las E/S tardan 10 ms cada una), mientras que B simplemente usa la CPU durante 50 ms y no realiza ninguna E/S. El planificador ejecuta primero a A, y a B después (Figura 7.8).

Supongamos que estamos intentando construir un planificador STCF. ¿Cómo debería tener en cuenta ese planificador el hecho de que A está dividido en 5 subtrabajos de 10 ms, mientras que B es solo una demanda de CPU de 50 ms? Claramente, ejecutar simplemente un trabajo y después el otro sin tener en cuenta la E/S no tiene mucho sentido.

Un enfoque común es tratar cada subtrabajo de 10 ms de A como un trabajo independiente. Por lo tanto, cuando el sistema se inicia, su elección está entre ejecutar un A de 10 ms o un B de 50 ms. Con STCF, la elección es clara: elegir el más corto, en este caso A. Después, cuando el primer subtrabajo de se haya completado, solo queda B y comienza su ejecución. Luego, se envía un nuevo subtrabajo de A,

que se adelanta a B y se ejecuta durante 10 ms. De esta manera se permite la **superposición**, con la CPU siendo utilizada por un proceso mientras espera que se complete la E/S de otro proceso; y por lo tanto, el sistema es mejor utilizado (véase la Figura 7.9).

Y así vemos cómo un planificador puede incorporar la E/S. Al tratar cada ráfaga de CPU como un trabajo, el planificador se asegura de que los procesos "interactivos" se ejecuten con frecuencia. Mientras esos trabajos interactivos realizan E/S, se ejecutan otros trabajos que requieren un uso intensivo de la CPU, aprovechando mucho más el procesador.

7.9 No Más Oráculo

Con un enfoque básico de E/S en su lugar, llegamos a nuestra última suposición: que el planificador conoce la duración de cada trabajo. Como dijimos antes, esta es probablemente la peor suposición que podríamos hacer. De hecho, en un SO de propósito general (como los que nos interesan), el SO generalmente sabe muy poco sobre la duración de cada trabajo. Entonces, ¿cómo podemos construir un enfoque que se comporte como SJF/STCF sin ese conocimiento a priori? Además, ¿cómo podemos incorporar algunas de las ideas que hemos visto con el planificador RR para que el tiempo de respuesta también sea bastante bueno?

7.10 Resumen

Hemos introducido las ideas básicas detrás de la planificación, y desarrollado dos tipos de acercamientos distintos. El primero ejecuta el trabajo más corto restante y, por lo tanto, optimiza el tiempo de entrega; el segundo alterna entre todos los trabajos y optimiza así el tiempo de respuesta. Por desgracia, ambos son malos donde el otro es bueno, un intercambio inherente que es común en los sistemas. También hemos visto cómo podemos incorporar E/S en el panorama, pero aún no hemos resuelto el problema de la incapacidad fundamental del SO para ver el futuro. En breve, veremos cómo superar este problema mediante la creación de un planificador que utiliza el pasado reciente para predecir el futuro. Este planificador se conoce como la **cola multinivel retroalimentada**, y es el tema del próximo capítulo.

7.11 Referencias

[B+79] "The Convoy Phenomenon" por M. Blasgen, J. Gray, M. Mitoma, T. Price. ACM Operating Systems Review, 13:2, Abril de 1979. *Quizás sea la primera referencia a convoyes, que se produce tanto en bases de datos como en SOs.*

[C54] "Priority Assignment in Waiting Line Problems" por A. Cobham. Journal of Operations Research, 2:70, páginas 70–76, 1954. *El artículo pionero en el uso de un enfoque SJF en la planificación de la reparación de máquinas.*

[K64] "Analysis of a Time-Shared Processor" por Leonard Kleinrock. Naval Research Logistics Quarterly, 11:1, paginas 59–73, Marzo de 1964. *Puede que sea la primera referencia al algoritmo de planificación round-robin; sin duda uno de los primeros análisis de dicho enfoque para programar un sistema de tiempo compartido.*

[CK68] "Computer Scheduling Methods and their Countermeasures" por Edward G. Coffman y Leonard Kleinrock. AFIPS '68 (Spring), Abril de 1968. *Una excelente primera introducción y análisis de una serie de disciplinas básicas de planificación.*

[J91] "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling" por R. Jain. Interscience, New York, Abril de 1991. *El texto estándar sobre la medición de sistemas informáticos. Una gran referencia para tu biblioteca, eso seguro.*

[O45] "Rebelión en la granja" por George Orwell. Secker and Warburg (London), 1945. *Un gran pero deprimente libro alegórico sobre el poder y sus corrupciones. Algunos dicen que es una crítica de Stalin y a la era de Stalin previa a la Segunda Guerra Mundial en la U.R.S.S; nosotros decimos que es una crítica a los cerdos.*

[PV56] "Machine Repair as a Priority Waiting-Line Problem" por Thomas E. Phipps Jr., W. R. Van Voorhis. Operations Research, 4:1, páginas 76–86, Febrero de 1956. *Trabajo de seguimiento que generaliza el enfoque SJF para la reparación de máquinas del trabajo original de Cobham; también postula la utilidad de un enfoque STCF en tal entorno. En concreto, "Hay ciertos tipos de trabajos de reparación, (. . .) que implican mucho desmantelar y recubrir el piso con tuercas y tornillos, que ciertamente no deben ser interrumpidos una vez realizados; en otros casos, sería desaconsejable continuar trabajando en un trabajo largo si uno o más trabajos cortos estuvieran disponibles (p. 81) ".*

[MB91] "The effect of context switches on cache performance" por Jeffrey C. Mogul, Anita Borg. ASPLOS, 1991. *Un buen estudio sobre cómo el cambio de contexto puede afectar al rendimiento de la caché; un problema menor en los sistemas actuales, donde los procesadores emiten miles de millones de instrucciones por segundo pero los cambios de contexto aún ocurren en el rango de tiempo de milisegundos.*

[W15] "You can't have your cake and eat it" por autores desconocidos.. Wikipedia (consultado en Diciembre de 2015).

`https://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it.`
La mejor parte de esta página es leer todos los modismos similares en otros idiomas. En tamil, no se puede "tener el bigote y beber la sopa".

7.12 Tarea (Simulación)

Este programa, `scheduler.py`, te permite ver cómo se desempeñan los diferentes planificadores con métricas de planificación tales como el tiempo de respuesta, el tiempo de entrega y el tiempo total de espera. Consultá el archivo README para obtener más detalles.

Preguntas

1. Calcular el tiempo de respuesta y el tiempo de entrega cuando se ejecutan tres trabajos de longitud 200 con los planificadores SJF y FIFO.
2. Ahora hacé lo mismo pero con trabajos de diferentes longitudes: 100, 200 y 300.
3. Ahora hacé lo mismo, pero además con el planificador RR y un segmento de tiempo de 1.
4. ¿Para qué cargas de trabajo ofrece SJF los mismos tiempos de entrega que FIFO?
5. ¿Para qué cargas de trabajo y longitudes de quantum ofrece SJF los mismos tiempos de respuesta que RR?
6. ¿Qué sucede con el tiempo de respuesta con SJF a medida que aumenta la duración del trabajo? ¿Podés usar el simulador para demostrar la tendencia?
7. ¿Qué sucede con el tiempo de respuesta con RR a medida que aumentan las longitudes de quantum? ¿Podés escribir una ecuación que dé el tiempo de respuesta en el peor de los casos, dados N trabajos?