Android, iOS, tiempos de respuestas y por qué nada es gratis en sistemas informáticos

Ricardo Galli

http://gallir.wordpress.com/2011/12/07/android-ios-tiempos-de-respuestas-y-por-que-nada-es-gratis-en-sistemas-informaticos/

Hace unas pocas horas escribí esta respuesta sobre *Por qué iOS es más fluido que Android* (con buen criterio, eliminaron la entrada). Obviamente, por cuestiones de longitud y la "respuesta rápida" que requiere un comentario, no me quedó todo lo completo que requiere el tema. Lo que me gustaría explicar daría para muchas horas de charlas. De hecho, enseño estos temas en mi asignatura de Sistemas Operativos (II), dedico al menos unas 12 hs de clase, y aún así no entramos en muchos detalles importantes. Pero intentaré resumirlo en este apunte, fundamentalmente para que se entiendan los problemas de arquitectura, y de cómo toda decisión que se tome en una arquitectura, lenguaje o programa tiene implicaciones positivas y negativas, siempre.

Lo básico

Apple tomó muchas decisiones técnicas con el objetivo de mejorar la "experiencia de usuario", pero por sí mismas serían inútiles -o más perjudiciales- si no van acompañadas de medidas de control estrictas. Es fundamental el papel de los controles de las aplicaciones en el App Store (¡ojo! no los justifico). Independientemente de otras consideraciones políticas, dado su simplicidad y soluciones *ad hoc*, el iOS sería muy fácil de "abusar" por los desarrolladores y aplicaciones.

Por el contrario, Android es una plataforma abierta que puede usar cualquier fabricante, que permite la instalación de cualquier aplicación. El control de aplicaciones del Market por parte de Google es prácticamente inexistente. Esto hace que no valgan soluciones *ad hoc*, obliga a implementar en el sistema operativo medidas de seguridad y control "canónicas", en el sentido que mantenga las condiciones fundamentales de todo sistema operativo de propósito general: eficiencia, equidad (*fairness*) y seguridad.

Cualquiera que haya estudiado la teoría y arquitectura de sistemas operativos (o que haya leído sobre el tema) entiende perfectamente que la eficiencia, equidad y seguridad son objetivos contradictorios. Mayor equidad o seguridad afectan negativamente a la eficiencia, y viceversa.

Hay otro problema importante. Se desea que los sistemas operativos de teléfonos actúen como sistemas de *tiempo real* también para las aplicaciones (ya tienen que serlos en otros aspectos, como la "radio" a interacción con protocolos de red de telefonía). Es decir, con límites precisos de "tiempo de respuesta": el tiempo que transcurre desde que ocurre un evento (como tocar la pantalla) hasta que se empieza a ejecutar al "gestor" de ese evento (la aplicación).

Los sistemas Unix (como Linux o Darwin) no son de tiempo real, su planificador de procesos (*scheduler*) es no-determinístico. No lo es por una sencilla razón: si se pretende una buena "multiprogramación" (llamada comunmente "multitarea") se deben

cumplir los objetivos de equidad y otros más complejos, como evitar los tiempos de espera muy largos (*starvation*). Esos dos objetivos impiden asegurar (formalmente) que el sistema sea de tiempo real. Además, los sistemas de tiempo real exigen que el tiempo de ejecución de cada ráfaga de ejecución de las aplicaciones (*CPU bursts*) críticas sea lo suficientemente breve para poder asegurar "tiempo real" a las demás aplicaciones (recordad que el scheduler debe ser determinístico, por ende no puede asegurar equidad).

En pocas palabras, si deseas comportamientos próximos al tiempo real necesitas relajar los requerimientos de un sistema de propósito general, y aumentar el control de las aplicaciones que se ejecutan en ese sistema.

¿Se entiende el balance y decisiones contradictorias que hay que tomar? En un sistema de multiprogramación, con el hardware equivalente (los dispositivos iOS y Android lo son), si se desea disminuir las *latencias* se deben relajar otras condiciones del planificación de procesador (por lo que se obtiene una "peor multitarea") y aumentar el control de aplicaciones (lo que genera otros problemas políticos, éticos, sociales y de negocio). Si pretende más apertura y libertad, se debe ser muy estricto con equidad y seguridad, lo que genera efectos negativos sobre el tiempo de respuesta.

Al final el tema de fondo no es [sólo] técnico, sino de otros aspectos más filosóficos: control vs apertura.

Creo que ya expliqué la base del problema técnico-filosófico, ya puedes dejar de leer si te agobian los detalles técnicos. Pero si te interesan, intentaré resumirlos en las cuatro partes que tienen más responsabilidad en los "tiempos de respuesta" de un sistema: las aplicaciones, el lenguaje y plataforma, la gestión de memoria, y el scheduler.

Las aplicaciones

El tipo de programación para móviles es esencialmente "dirigida por eventos". Cuando el usuario toca la pantalla se llama la función que debe responder a ese evento (en Android se llaman "actividades"). La recomendación es que la aplicación no necesite mucha CPU ni tiempo para generar la respuesta, especialmente en los menús principales. Por supuesto, a este nivel, todo depende de los programadores de la aplicación, y supongo que estos son los mayores culpables de la "reacción lenta" de los menús cuando se interactúa con una aplicación.

Las aplicaciones también pueden tener otros malos comportamientos, por ejemplo consumir mucha CPU (aún cuando están en *background* o como servicio), no sólo molesta a todos los demás procesos, también aumenta el consumo de batería y la temperatura.

¿Cómo solucionar este problema? O se hace un control estricto de cada aplicación, o se implanta un scheduler más sofisticado y genérico que puede tener algunas aristas.

Ya sabéis qué decisión se tomó en Apple y cuál en Google.

El lenguaje y plataforma

iOS y Android difieren mucho en este aspecto. iOS usa el mismo lenguaje estándar del Mac OS X, el Objective-C. Los programas se compilan directamente en código ejecutable para el procesador. En cambio Android optó por usar Java, que se convierte a un lenguaje intermedio ejecutado por una máquina virtual optimizada (similar la máquina virtual de Java, pero es diferente), Dalvik.

Obviamente el rendimiento entre ambas es muy diferente, el código ejecutable siempre será más eficiente que otro que no se ejecuta directamente en el procesador. Pero esto no significa que los diseñadores de Android sean unos imbéciles, de nuevo, fueron decisiones estratégicas que están muy por encima de temas técnicos, aunque terminan afectando dramáticamente a este último.

Si el sistema operativo se va a ejecutar siempre sobre una única arquitectura, como hizo históricamente Apple (aunque tuvo dos migraciones importantes, exitosas pero dolorosas y muy molestas para los desarrolladores), la opción natural es generar directamente el ejecutable. Además, Apple licenció el diseño de ARM [*] y compró una empresa de diseño de chips (Intrinsity) para que se encargue del diseño de sus procesadores y no depender ni de fabricantes externos (antes lo hacía Samsung).

Por otro lado, la intención de Android era entrar a competir en el mercado con un sistema libre/abierto y multiplataforma (después de varios intentos fallidos, como OpenMoko), por lo que era preferible un sistema que no obligase a compilar una aplicación para cada tipo de procesador. Si no, estaría condenada al fracaso. Para solucionar el problema de la eficiencia diseñaron desde cero el Dalvik, que al usar el lenguaje Java ya tenía a su disposición muchas herramientas (como Eclipse), librerías de desarrollo (como las clases estándar de Java), y una comunidad enorme de progamadores (me abstengo de opinar sobre Java, pero afortunadamente Android lo ha mejorado muchísimo con su API y *framework* de programación).

Aunque Java ya está preparado para *multithreading*, por cuestiones de seguridad los diseñadores de Android prefirieron, muy acertadamente, ejecutar cada aplicación como un proceso diferente y completamente aislado. Dejaron esta parte de la gestión de procesos al núcleo Linux, que es muy eficiente y está más que probado. Con esta arquitectura de máquinas virtuales replicadas en procesos independientes, se generan otras ineficiencias: cada proceso debe cargar su máquina virtual, y éste debe cargar después todas las clases estándar de Java que hacen falta para la aplicación. Esto haría que el tiempo de arranque de cada programa fuese inaceptable, y han recurrido a un truco muy guapo para solucionarlo (por eso digo que la arquitectura de Android es una obra impresionante de ingeniería informática): el *zygote*.

El kernel Linux implementa un mecanismo llamado *copy-on-write* (COW) que permite que los procesos hijos (creados con el fork()) compartan memoria con el padre (por eso la creación de procesos es muy rápida de Linux). Cuando un proceso crea un hijo, éste reutiliza la misma memoria y tablas de páginas del padre, pero todas las páginas se marcan como “sólo lectura”. El hijo puede comenzar a ejecutarse muy rápidamente. Cuando uno de los procesos intenta escribir sobre una página compartida, se genera una interrupción de error (*trap*), el sistema operativo coge el control, verifica que se trata de memoria COW, asigna una nueva página, copia el contenido de la página afectada,

modifica la tabla de páginas de uno de los procesos para apuntar a esta nueva, y permite la continuación de ambos procesos. Además de la rapidez con que puede crear y ejecutar procesos, se ahorra mucha memoria RAM, por ejemplo, el código ejecutable y las librerías comunes siempre se comparten (nunca se modifican).

¿Cómo aprovecha Android este mecanismo? Crea un proceso inicial, el *zygote*, que carga la máquina virtual y las librerías estándares de Java y del API de Android. Cada nueva aplicación que se arranca es hija de este zygote, et voilà, ya ganaron eficiencia y se ahorra muchísima memoria (Chrome y Chromium usan la misma técnica para abrir pestañas y ventanas rápidamente).

¿Es o no es una obra maestra de ingeniería? Claro que sí, se han tomado todo ese trabajo para compensar el problema de eficiencia y seguridad de una máquina virtual. Por eso, cuando pienso que por debajo hay una máquina virtual ejecutando el código, me maravillo con la velocidad de ejecución de las aplicaciones en Android.

Por supuesto, esta arquitectura con memoria virtual tiene otros efectos negativos a la hora de medir los tiempos de respuesta: en la gestión de memoria y cache, que trato más adelante.

[*] En la biografía de Steve Jobs cuentan como despuésde evaluar usar Intel para los iPad, Jobs le dijo muy cabreado a los ejecutivos de Intel algo así como: “son muy buenos para procesadores de alto rendimiento, pero sois unos inútiles para procesadores de bajo consumo”.

La gestión de memoria del núcleo

Los procesadores que usan iOS y Android son similares en cuanto a su gestión de memoria (también similares a los ordenadores que usáis). Ambos gestionan memoria virtual con paginación. Los procesos no generan direcciones físicas de RAM, sino direcciones lógicas que son convertidas a direcciones físicas en cada ejecución por el módulo conocido como *Memory Manager Unit* (MMU). Cada proceso tiene sus propias tablas de página que mantienen la relación de número de página del proceso (es el índice a la tabla) a ubicación en la memoria RAM. Los procesadores ARM permiten páginas de 4KB a 64KB (lo usual son 4KB), así que cada aplicación tiene miles de páginas, y por lo tanto tablas de miles de entradas (en ARM hay tablas de dos niveles, en la arquitectura Intel o AMD, cuatro niveles).

Cada entrada de la tabla almacena propiedades de la página: lectura (r), escritura (w), ejecución (x), accedida (a, o *acces bit*), modificada (d, o *dirty bit*), cargada en memoria (p, *present bit*), etc. Por esta razón las tablas se mantienen en la memoria RAM, lo que genera otro problema ¿cómo hace el MMU para no tener que acceder dos veces a la memoria -primero a la tablas de página y luego a la dirección que se pretende acceder-? Sin un mecanismo que lo solucione, el tiempo efectivo de acceso a la memoria sería al menos el doble de lo que permite la memoria RAM (o sea, una patata de lento).

Para solucionar este problema se usan sistemas de cache que almacenan en registros del procesador un conjunto de entradas de las tablas: el *translation-lookaside-buffer* (TLB). El TLB de los ARM suelen tener 128 entradas, cada una de ellas es una copia idéntica de la entrada de la tabla de páginas. Se intenta tener en el TLB las entradas más usadas,

así, cada vez que el MMU encuentra la entrada en el TLB puede acceder directamente a la memoria (*hit*), si no, tiene que descartar una entrada y cargar la que hace falta (*fault*). El *hit ratio* es una indicación de la velocidad de acceso efectivo a memoria, viene dado por el procesador y el algoritmo de sustitución que se use, a mayor número de entradas en el TLB, el *hit ratio* sube.

Cada vez que el scheduler selecciona otra aplicación para ejecutar se produce un *cambio de contexto*, entre otras cosas significa que las entradas del TLB deben ser invalidadas (ya no sirven para el nuevo proceso, por seguridad tampoco debe poder usar esos datos), además las entradas que han sido modificadas por el procesasor (por ejemplo, que una una página fue accedida o modificada) deben ser copiadas a la tabla original en memoria (*flush*). Por eso se dice que los cambios de contexto son "caros", y es muy imporante las decisiones que toma el scheduler para minimizar los cambios de contexto "inútiles" (¡pobre el scheduler!, las cosas que tiene que tener en cuenta para que lo usuarios no se quejen porque el audio pega saltos).

Un proceso con máquina virtual como Android mete mucha más presión al TLB (y por ende baja el *hit ratio*) que un proceso que ejecuta código nativo. Para ejecutar una función equivalente en máquina virtual se necesita más memoria, la del código del intérprete, la memoria usado por éste, más el código en lenguaje intermedio y la memoria que necesita el programa. No sólo eso, prácticamente cada ejecución del código intermedio implica cambios en las variables del propio programa, también en las de la máquina virtual, forzando a que en cada cambio de contexto sean más las entradas del TLB que deben ser copiadas a la RAM. O sea, los cambios de contexto son también más caros.

Lo mismo que pasa con el TLB (recordemos que es un tipo específico de cache), pasa con la memoria de cache de RAM en el procesador, la máquina virtual mete más presión, por lo que también bajan sus *hit ratios*.

De nuevo, una decisión de negocio (y política y filosófica), genera mcuhas complicaciones técnicas que hay que resolver. Si uno las tiene en cuenta, empieza a dejar de pensar en términos de *fanboy* y se cuestiona muchas cosas. Entre otras, lo eficiente que tiene que ser Linux para gestionar esta complejidad adicional sin que se note demasiado la diferencia.

El scheduler

Se llama scheduler al módulo del sistema operativo que selecciona cuál será el siguiente proceso a ejecutar. A cada proceso se le asigna un tiempo máximo de ejecución: el cuanto (o *quantum*). Si al proceso se le acaba el cuanto, el sistema operativo se apropia de la CPU (por eso se llaman apropiativos o *preemptive*), y llama al scheduler para seleccionar el siguiente de la lista de procesos que esperan para ejecutar (este mecanismo se denomina *round robin*). El scheduler de Linux (como la gran mayoría) no son determinísticos, significa que a priori no se puede conocer la secuencia de ejecución de procesos, por lo que no se puede asegurar analítica o formalmente que los procesos se ejecuten en un tiempo máximo predeterminado (si éste es suficientemente pequeño).

No todos los procesos tienen los mismos requerimientos, aquellos interactivos, o de reproducción multimedia necesitan menores tiempo de respuesta que un servicio en *background*, o que consume mucha CPU. Para eso se usan las prioridades, cada proceso tiene una prioridad inicial (por ejemplo "multimedia", "normal", "de fondo", etc.), luego el *scheduler* la ajusta dinámicamente (prioridad dinámica) dependiendo del comportamiento del proceso y las necesidades de los otros. Por eso, el comportamiento de un sistema que use prioridades depende del comportamiento global de todos los procesos, estos no se pueden predecir, y de allí que sea no determinístico, y que se generen tiempos de respuesta variables.

El scheduler además tiene que tomar en cuenta muchos factores adicionales (p.e., minimizar los cambios de contexto), y evitar que se produzcan esperas muy largas (*starvation*) que se pueden producir porque procesos con mayor prioridad están cogiendo siempre la CPU y no dejan ejecutar a otros de menor prioridad. Imaginaros que tenéis el reproducto de música de fondo, y que estáis por sacar una foto. La aplicación de foto podría estar consumiendo el 100% de CPU para actualizar la imagen, por lo que produciría cortes en la música, insoportablemente molesto ¿no? Pues es el pobre scheduler el que tiene que evitar estos problemas, tomando decisiones muy rápidas, cientos de veces por segundo.

Para evitar estas esperas tan largas en los casos extremos (*corner cases*) como el que acabo de decir, el scheduler mantiene dos colas, la *activa* y la *inactiva*. Siempre asigna procesos desde la cola activa. A cada proceso se le asigna un tiempo máximo (además del cuanto) que puede estar en la cola activa (el *timeslice*, por ejemplo 500 ms), cuando consume todo su tiempo se le mueve a la cola inactiva. Cuando ya no queda nadie en la cola activa, el scheduler cambia, la activa pasa a ser la inactiva, y viceversa. Esto aumenta aún más el efecto "no determinístico" del scheduler, y puede producir saltos o latencias elevadas si el procesador está temporalmente sobrecargado.

Para evitar estas latencias que molestan a la interactividad del usuario se pueden tomar varias medidas ad hoc, por ejemplo subir la prioridad al máximo a la aplicación que está activa (así hacía Windows), pero esto genera otros problemas: ¿si la aplicación activa abusa y no deja de usar la CPU? (un truco obvio de programador para que su aplicación parezca que es más rápida y eficiente ¿no?) ¿y si hay procesos en background que necesitan la CPU -por ejemplo un SMS, o responderá la red antes que pase el timeout y esperan demasiado tiempo?

Pues es muy complicado, y cualquier "truco" que se ponga al scheduler deja la puerta abierta para que los programadores abusen… salvo que hagas un control exhaustivo de cada aplicación que se vaya instalar en el teléfono. ¿Se entiende por qué Apple puede recurrir a trucos ad hoc y evitar al mismo tiempo el abuso de las aplicaciones?

Si por el contrario renuncias a hacer ese control, no queda más remedio que el scheduler haga su trabajo, como en cualquier sistema operativo de uso genérico. Y que la solución adoptada sea lo suficientemente buena, simple, genérica y probadamente segura. ¿Se entiende por qué la solución en el Linux de Android es más difícil de encontrar? o ¿por qué le llamé una "solución canónica?

¿Como puede Android mejorar los tiempos de respuesta?

Evidentemente hay dos caminos obvios.

Por un lado, mejorar paulatinamente el scheduler (si es que allí reside el problema). No es una cuestión de un día, ni un año, en Linux se tardó años en encontrar un planificador que funcione tan bien como la actual para sistemas interactivos (el scheduler CFQ - http://en.wikipedia.org/wiki/Completely_Fair_Scheduler).

La otra parte, la gestión de memoria, depende en gran medida del apoyo del hardware. Un procesador con más capacidad de cache y TLB mejoraría mucho, pero también tiene su coste: más transistores, más consumo, más temperatura, mayor tamaño. A medida que se mejore el proceso de fabricación (más transistores en el mismo chip), seguramente incluirán TLB y cache con más capacidad. Lo mismo ocurre con el aumento de la potencia bruta y el aumento de cores (aunque creo que afectan en menor medida a los tiempos de respuesta de la interfaz).

Tampoco me extrañaría que el código de los drivers de la pantalla sea de baja calidad (como suelen ser todos los drives, comparados con el núcleo del sistema operativo, se hacen bajo presión, y la prioridad es el hardware, no el SO) y encuentren *big locks* que no tocan.

De todas formas, los tiempos irán mejorando paulatinamente hasta que no podamos detectar diferencias en las latencias de la interactividad. Al mismo tiempo, iOS tendrá que ir incluyendo mejores capacidad de "multitarea" (como ya pasó desde que salió la primera versión de iOS, y seguramente irá reduciendo los controles de calidad que hace a cada aplicación).

Supongo que os dais cuenta de la complejidad técnica de estos "cacharritos", y que las decisiones de negocio imponen muchas restricciones y problemas al desarrollo del software, lo que obliga a los ingenieros a aplicarse a fondo. No es que los ingenieros de iOS son unos genios, y los de Android unos inútiles, los requerimientos y *grados de libertad* son diferentes.

Ley informática: optar por la apertura tiene *costes* técnicos iniciales, optar por el software libre tiene *costes* técnicos iniciales, pero producir una plataforma totalmente bajo control también tiene *costes* (sociales), y a más largo plazo.