Elementos de diseño del sistema de Memoria Virtual de FreeBSD

Resumen

El título es sólo una forma elegante de decir que voy a intentar describir la enchilada de la Memoria Virtual, espero que de un modo que todo el mundo pueda seguir. Durante el último año me he concentrado en un número de subsistemas principales del kernel de FreeBSD, con los subsistemas de Memoria Virtual e Intercambio siendo los más interesantes y NFS siendo "una tarea necesaria". Reescribí sólo pequeñas porciones del código. En el área de Memoria Virtual la única reescritura importante que he hecho es el subsistema de intercambio. La mayor parte de mi trabajo fue de limpieza y mantenimiento, con tan sólo reescrituras moderadas de código y sin ajustes algorítmicos importantes en el subsistema de Memoria Virtual. El grueso de la base teórica del subsistema de Memoria Virtual permanece sin cambios y mucho del crédito del esfuerzo de modernización en los últimos años es para John Dyson y David Greenman. Como no soy un historiador como Kirk no intentaré etiquetar todas las características con nombres de personas, ya que me equivocaría irremediablemente.

Tabla de contenidos

1. Introducción

Antes de avanzar con el diseño real dediquemos un poco de tiempo a la necesidad de mantener y modernizar cualquier base de código de larga duración. En el mundo de la programación, los algoritmos tienen a se más importantes que el código y es precisamente debido a las raíces académicas de BSD que una gran parte de la atención se puso desde el comienzo en el diseño algorítmico. Prestar más atención al diseño generalmente lleva a una base de código limpio y flexible que puede ser modificado fácilmente, extendido o reemplazado a lo largo del tiempo. Aunque BSD es considerado por alguna gente como un sistema operativo"viejo", aquellos de nosotros que trabajamos en él solemos verlo más como una base de código"madura" la cual tiene

varios componentes modificados, extendidos, o reemplazados con código moderno. Ha evolucionado, y FreeBSD está a la vanguardia independientemente de cómo de viejo sea parte del código. Es importante hacer esta distinción y que mucha gente pasa por alto. El mayor error que puede cometer un programador es no aprender de la historia, y es precisamente este error el que han cometido muchos otros sistemas operativos modernos. Windows NT® es el mejor ejemplo, y las consecuencias han sido nefastas. Linux también comete este error hasta cierto punto—lo suficiente para que nosotros la gente de BSD hagamos pequeñas bromas de vez en cuando, por lo menos. El problema de Linux es simplemente la falta de experiencia y de una historia contra la que comparar ideas, un problema que está siendo tratado rápidamente por la comunidad Linux de la misma forma que ha sido tratado en la comunidad BSD—mediante el desarrollo continuo de código. La gente de Windows NT®, por otro lado, repiten los mismos errores solucionados por UNIX® hace décadas y pasan años arreglándolos. Una y otra vez. Sufren un caso severo de "no diseñado aquí" y "siempre tenemos la razón porque nuestro departamento de marketing así lo dice". Tengo poca tolerancia hacia cualquiera que no puede aprender de la historia.

Mucha de la aparente complejidad del diseño de FreeBSD, especialmente en el subsistema de Memoria Virtual/Intercambio, es un resultado directo de tener que resolver serios problemas de rendimiento que ocurren bajo condiciones variadas. Estos problemas no se deben aun mal diseño algorítmico sino que surgen de factores ambientales. En cualquier comparación directa entre plataformas, estos problemas se hacen más evidentes cuando los recursos del sistema empiezan a sufrir estrés. Como describo en el subsistema de Memoria Virtual/Intercambio de FreeBSD el lector siempre debería tener en mente dos puntos:

1. El aspecto más importante del diseño de rendimiento es lo que se conoce como "Optimización del Camino Crítico". Es común que las optimizaciones de rendimiento inflen algo el código con el fin de mejorar el rendimiento del camino crítico.
2. Un diseño sólido, generalizado tiene mejor rendimiento a largo plazo que un diseño altamente optimizado. Mientras que un diseño generalizado puede terminar siendo más lento que un diseño altamente optimizado cuando se implementan inicialmente, el diseño generalizado tiende a ser más fácil de adaptar a condiciones cambiantes y el diseño altamente optimizado termina siendo desechado.

Cualquier base de código que sobrevivirá y será mantenible durante años debe por lo tanto ser diseñada adecuadamente desde el comienzo incluso si tiene algo de coste en rendimiento. Hace veinte años la gente todavía discutía si la programación en ensamblador era mejor que programar en un lenguaje de alto nivel porque producía código que era diez veces más rápido. Hoy, la falibilidad de ese argumento es obvio — de modo paralelo al diseño algorítmico y la generalización de código.

2. Objetos de Memoria Virtual

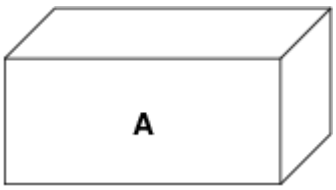
La mejor manera de empezar describiendo el sistema de Memoria Virtual de FreeBSD is mirarlo desde la perspectiva de un proceso de usuario. Cada proceso de usuario ve una espacio de direcciones de Memoria Virtual único, privado y contiguo que contiene diversos tipos de objetos de memoria. Estos objetos tienen diversas características. Código de programa y datos de programa son de forma efectiva un solo fichero mapeado en memoria (el fichero binario que se está ejecutando), pero el código del programa es de solo lectura mientras que los datos de programa son

copy-on-write. El BSS del programa es sólo memoria asignada y rellenada con ceros bajo demanda, llamada rellenado de página cero bajo demanda. Ficheros arbitrarios pueden ser mapeados en memoria en el espacio de direcciones también, que es como funciona el mecanismo de librerías compartidas. Dichos mapeos pueden requerir modificaciones para permanecer privados al proceso que los realiza. La llamada al sistema fork añade una nueva dimensión a al problema de la gestión de la Memoria Virtual a añadir a la complejidad ya existente.

Una página de datos binarios de un programa (que es una página copy-on-write básica) ilustra esta complejidad. Un programa binario contiene una sección de datos preinicializados que es inicialmente mapeado directamente por el fichero del programa. Cuando un programa se carga en el espacio de direcciones de Memoria Virtual del proceso, este área es inicialmente mapeado en memoria y respaldado por el programa binario mismo, permitiendo al sistema de Memoria Virtual liberar/reutilizar la página y después cargarla de nuevo desde el binario. Sin embargo, en el momento en que un proceso modifica estos datos, el sistema de Memoria Virtual debe hacer una copia privada para ese proceso. Puesto que la copia privada ha sido modificada, el sistema de Memoria Virtual podría no ser capaz de liberarla, porque no hay forma de restaurarla posteriormente.

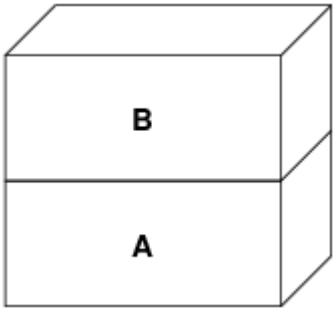
Notarás inmediatamente que lo que originalmente era un simple mapeo de un fichero se ha convertido en algo más complejo. Los datos pueden ser modificados página a página mientras que el mapeo de ficheros engloba varias páginas a la vez. La complejidad aumenta más cuando un proceso se bifurca. Cuando un proceso se bifurca, el resultado son dos procesos—cada uno con su propio espacio de direcciones privado que incluye cualquier modificación hecha por el proceso original antes de la llamada a `fork()`. Sería tonto para el sistema de Memoria Virtual hacer una copia completa de los datos en el momento de llamar a `fork()` porque es bastante posible que al menos uno de los dos procesos solamente necesite leer de esa página a partir de este momento, permitiendo así que se use la página original. Lo que era una página privada se ha convertido en una página copy-on-write de nuevo, puesto que cada proceso (padre y hijo) espera que sus propias modificaciones personales después del fork permanezcan privadas para ellos y que afecten al otro.

FreeBSD gestiona todo esto con un modelo de Objetos de Memoria Virtual en capas. El fichero del programa binario original termina siendo la capa de Objetos de Memoria Virtual más baja. Una capa copy-on-write se sitúa encima de ella para mantener aquellas páginas que han sido copiadas del fichero original. Si el programa modifica una página de datos que pertenece al fichero original el sistema de Memoria Virtual recibe un fallo de página y hace una copia de la página en la capa superior. Cuando un proceso bifurca, se empujan nuevas capas de Objetos de Memoria Virtual. Esto puede cobrar más sentido con un ejemplo bastante básico. Un `fork()` es una operación común para cualquier sistema *BSD, así que este ejemplo considerará un programa que arranca, y bifurca. Cuando el proceso arranca, el sistema de Memoria Virtual crea una capa de objetos, llamémosla A:

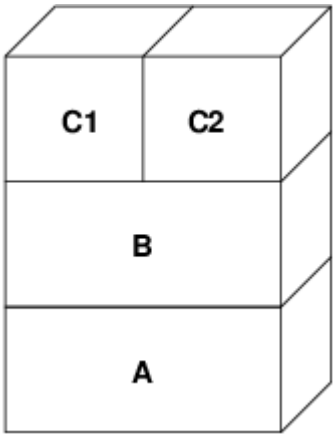


A representa el fichero—las páginas pueden ser paginadas hacia o desde el medio físico del fichero según sea necesario. Paginar desde el disco es razonable para un programa, pero en realidad no queremos paginar de vuelta y sobrescribir el ejecutable. Por tanto, el sistema de Memoria Virtual

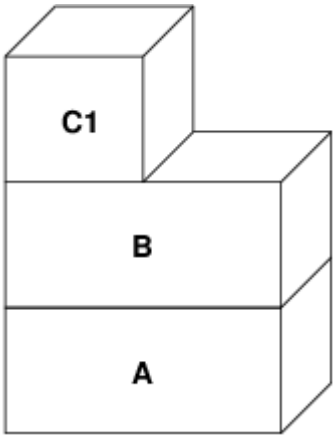
crea una segunda capa, B, que estará respaldada físicamente por espacio de intercambio:



En la primera escritura a una página después de esto, se crea una nueva página en B, y su contenido es inicializado desde A. Todas las páginas en B pueden ser paginadas a o desde el dispositivo de intercambio. Cuando el programa bifurca, el sistema de Memoria Virtual crea dos capas de objetos nuevas—C1 para el padre, y C2 para el hijo—que descansan sobre B:



En este caso, digamos que una página en B es modificada por el proceso padre original. El proceso recibirá un fallo de copy-on-write y duplicará la página en C1, dejando la página original en B sin tocar. Ahora, digamos que la misma página en B es modificada por el proceso hijo. El proceso recibirá un fallo de copy-on-write y duplicará la página en C2. La página original en B está ahora completamente oculta ya que tanto C1 como C2 tienen una copia y B podría teóricamente ser destruida si no representa un fichero "real"; sin embargo, este tipo de optimización no es trivial de hacer porque es muy fina. FreeBSD no hace esta optimización. Ahora, supón (como suele ser el caso) que el proceso hijo hace un `exec()`. Su espacio de direcciones actual es habitualmente remplazado por un nuevo espacio de direcciones que representa un nuevo fichero. En este caso, la capa C2 es destruida:



En este caso, el número de hijos de B ha bajado a uno, y todos los accesos a B van ahora a través de C1. Esto significa que B y C1 pueden colapsarse juntas. Cualquier página en B que también existe en C1 se borran de B durante el colapso. Por lo tanto, incluso aunque la optimización en el paso anterior no se pudo hacer, podemos recuperar las páginas muertas bien cuando el proceso sale o cuando llama a `exec()`.

Este modelo crea un número de problemas potenciales. El primero es que puedes terminar con una pila de Objetos de Memoria Virtual relativamente profunda que puede tener un coste de tiempo de escaneo y de memoria cuando recibes un fallo. Capas muy profundas pueden ocurrir cuando los procesos se bifurcan y se bifurcan de nuevo (en el padre o en el hijo). El segundo problema es que puedes terminar con páginas muertas, inaccesibles en lo profundo de la pila de Objetos de Memoria Virtual. En nuestro último ejemplo si tanto los el proceso padre como el hijo modifican la misma página, ambos obtienen su propia copia privada de la página y la página original en B ya no es accesible por nadie. Esa página en B puede ser liberada.

FreeBSD soluciona el problema de capas profundas con una optimización especial llamada "Caso de Todo Sombreado". Este caso ocurre si C1 o C2 generan suficientes fallos COW como para sombrear (ocultar) todas las páginas en B. Digamos que C1 lo consigue. C1 puede ahora puentear B completamente, así que en lugar de tener C1→B→A y C2→B→A ahora tenemos C1→A y C2→B→A. Pero mira lo que ha pasado también—ahora B tiene sólo una referencia (C1), así que podemos colapsar B y C2 juntas. El resultado final es que B se borra completamente y tenemos C1→A y C2→A. Habitualmente el caso es que B contendrá un gran número de páginas y ni C1 ni C2 serán capaces de ocultarla completamente. Si bifurcamos de nuevo y creamos un conjunto de capas D, sin embargo, es mucho más probable que una de las capas de D eventualmente sea capaz de ocultar el conjunto mucho menor representado por C1 o C2. La misma optimización funcionará en cualquier punto del grafo y el resultado total de esto es que incluso en una máquina con muchas bifurcaciones las pilas de Objetos de Memoria Virtual tienen a no ser mucho más profundas de 4. Esto es verdad tanto para el padre como para los hijos y es así tanto si el padre hace la bifurcación como si los hijos bifurcan en cascada.

El problema de la página muerta todavía existe en el caso en el que C1 o C2 no ocultan completamente B. Debido a otras optimizaciones este caso no es demasiado problema y simplemente permitimos que haya páginas muertas. Si el sistema se queda sin memoria las intercambiará a disco, utilizando un poco de espacio de intercambio, pero eso es todo.

La ventaja del modelo de Objetos de Memoria Virtual es que `fork()` es extremadamente rápido, ya que no se necesita realizar una copia real de datos. La desventaja es que puedes construir un conjunto de capas de Objetos de Memoria Virtual relativamente complejo que haga un poco más lento el manejo de fallos de página, y que tienes que gastar memoria en la gestión de las estructuras de los Objetos de Memoria Virtual. Las optimizaciones que hace FreeBSD demuestran que reducen los problemas lo suficiente de forma que pueden ser ignorados, eliminando prácticamente la desventaja.

3. Capas de Intercambio

Las páginas de datos privadas se crean como páginas copy-on-write o rellenas con ceros. Cuando se hace un cambio, y por lo tanto una copia, el objeto de respaldo original (normalmente un fichero) ya no puede ser utilizado para guardar una copia de la página cuando el sistema de Memoria

Virtual necesita reutilizarla para otros fines. Aquí es donde aparece el Intercambio. El Intercambio se asigna para crear almacenamiento de respaldo para memoria que de otra forma no la tendría. FreeBSD asigna la estructura de gestión del intercambio para un Objeto de Memoria Virtual solo cuando se necesita realmente. Sin embargo históricamente, la estructura de gestión del intercambio ha tenido problemas:

- En FreeBSD 3.X la estructura de gestión de intercambio preasigna un array que engloba todo el objeto que requiere almacenamiento de respaldo de intercambio—incluso si solo unas pocas páginas de ese objeto están respaldadas en el área de intercambio. Esto crea un problema de fragmentación de la memoria del núcleo cuando se mapean objetos grandes, o cuando procesos con tamaños de ejecución grandes (RSS) bifurcan.
- Además, para llevar la cuenta del espacio de intercambio, una "lista de huecos" es mantenida en la memoria del núcleo, y esta tiende a fragmentarse de forma severa también. Puesto que la "lista de huecos" es una lista lineal, el rendimiento de asignación y liberación de intercambio es de un orden subóptimo de O(n) por página.
- Requiere que se lleven a cabo asignaciones de memoria del núcleo durante el proceso de liberación de espacio de intercambio, y eso crea problemas de bloqueo por baja memoria.
- El problema se exacerba debido a los huecos creados por el algoritmo de entrelazado.
- Además, el mapa de bloques de intercambio se puede fragmentar fácilmente dando como resultado asignaciones no contiguas.
- La memoria del núcleo se debe asignar al vuelo para las estructuras adicionales de gestión de intercambio cuando se escribe en el área de intercambio.

De esa lista se hace evidente que había mucho margen de mejora. Para FreeBSD 4.X, reescribí completamente el subsistema de intercambio:

- Las estructuras de gestión de intercambio se asignan mediante una tabla has en lugar de un array lineal dándoles un tamaño de asignación fijo y mucha mayor granularidad.
- En lugar de utilizar una lista enlazada linear para llevar la cuenta de las reservas de espacio de intercambio, ahora usa un mapa de bits de bloques de intercambio dispuestos en una estructura tipo árbol radix con anotaciones sobre el espacio libre en las estructuras de nodos del radix.
- El mapa de bits entero para el árbol radix también se preasigna para evitar tener que asignar memoria del núcleo durante operaciones de intercambio con un nivel crítico de memoria baja. Después de todo, el sistema tiende a utilizar intercambio cuando está bajo en memoria de forma que deberíamos evitar asignar memoria del núcleo en esas situaciones para evitar potenciales bloqueos.
- Para reducir la fragmentación el árbol radix es capaz de asignar de una sola vez grandes trozos contiguos, saltándose pequeños trozos fragmentados.

No realicé el paso final de tener un "puntero de anotaciones para las asignaciones" que recorrería una porción del espacio de intercambio según se hicieran las asignaciones para así garantizar asignaciones contiguas o al menos localidad de referencia, pero aseguré que esa condición no podría darse.

4. Cuando liberar una página

Como el sistema de Memoria Virtual usa toda la memoria disponible para cachear disco, normalmente hay pocas páginas que estén realmente libres. El sistema de Memoria Virtual depende de su habilidad para adecuadamente escoger las páginas que no están en uso para reutilizarlas en nuevas asignaciones. Seleccionar las páginas óptimas para liberar es posiblemente la función más importante que cualquier sistema de Memoria Virtual puede realizar porque si la elección no es buena, el sistema de Memoria Virtual puede verse forzada a recuperar páginas de disco innecesariamente, degradando seriamente el rendimiento del sistema.

¿Cuánto trabajo extra estamos dispuestos a sufrir en el camino crítico para evitar liberar la página equivocada? Cada decisión errónea que hacemos costará cientos de miles de ciclos de CPU y una parada notable de los procesos afectados, así que estamos dispuestos a soportar una cantidad significativa de trabajo extra para estar seguros que se escoge la página adecuada. Por esto es por lo que FreeBSD tiende a superar en rendimiento a otros sistemas cuando se estresan los recursos de memoria.

El algoritmo que determina la página libre se construye en base al histórico de uso de las páginas de memoria. Para adquirir este histórico, el sistema se aprovecha de la característica del bit de página utilizada que la mayoría del hardware de tablas de página posee.

En cualquier caso, el bit de página utilizada se blanquea y en algún momento posterior el sistema de Memoria Virtual se encuentra con la página de nuevo y ve que el bit de página utilizada ha sido marcado. Esto indica que la página todavía se está utilizando activamente. Si el bit está blanqueado eso indica que la página no se usa activamente. Mediante el chequeo periódico de este bit, se desarrollo (en forma de contador) un histórico de uso . Cuando posteriormente el sistema de Memoria Virtual necesita liberar algunas páginas, examinar este histórico se convierte en la piedra de toque para determinar la mejor página candidata para reutilizar.

Para esas plataformas que no tienen esta característica, el sistema en realidad emula un bit de página utilizada. Desmapea o protege una página, forzando un fallo de página si ésta es accedida de nuevo. Cuando se maneja el fallo de página, el sistema simplemente marca la página como usada y desprotege la página de forma que puede ser utilizada. Aunque realizar este fallo de página tan solo para determinar si una página está siendo usada puede parecer una proposición cara, es mucho menos cara que reutilizar la página para otro propósito para darse cuenta después de que otro proceso la necesita y tener que ir al disco.

FreeBSD utiliza varias colas de páginas para refinar aún más la selección de páginas a reutilizar así como para determinar cuando se deben llevar las páginas sucias a su almacenamiento de respaldo. Puesto que las tablas de páginas en FreeBSD son entidades dinámicas, cuesta virtualmente nada desmapear una página del espacio de direcciones de cualquier proceso que la esté usando. Cuando se ha escogido una página candidata basándose en el contador de página utilizada, esto es precisamente lo que se hace. El sistema debe distinguir entre páginas limpias que pueden en teoría ser liberadas en cualquier momento, y páginas sucias que deben ser escritas primero en el almacenamiento de respaldo antes de ser reutilizadas. Cuando se encuentra una página candidata se mueve a la cola inactiva si está sucia, o a la cola de caché si está limpia. In algoritmo separado que se bajas en el ratio de páginas sucias respecto de las limpias determina cuándo se tienen que escribir a disco las páginas sucias de la cola inactiva. Una vez hecho esto, las páginas escritas se

mueven de la cola inactiva a la cola de caché. En este punto, las páginas en la cola de caché todavía pueden ser reactivadas por un fallo de Memoria Virtual con un coste relativamente bajo. Sin embargo, las páginas de la cola de caché se consideran como "inmediatamente liberables" y serán reutilizadas de modo LRU (Usada Menos Recientemente) cuando el sistema necesita asignar nueva memoria.

Es importante señalar que el sistema de Memoria Virtual de FreeBSD intenta separar páginas limpias y sucias para expresar la razón de evitar la escritura innecesaria de páginas sucias (que come ancho de banda de E/S), y tampoco mueve de forma gratuita páginas entre distintas colas de páginas cuando el sistema de memoria no está bajo estrés. Este es el motivo por el que verás algunos sistemas con contadores de cola de caché muy bajos y contadores de cola de páginas activa altos cuando se ejecuta el comando `systat -vm`. Según el sistema de Memoria Virtual va sufriendo más estrés, hace un gran esfuerzo por mantener varias colas de páginas en los niveles que determina que son más efectivos.

Durante años ha circulado una leyenda urbana acerca de que Linux hacía un mejor trabajo que FreeBSD evitando escribir en intercambio, pero de hecho esto no es cierto. Lo que ocurría en realidad era que FreeBSD estaba llevando a intercambio de forma proactiva páginas no utilizadas para hacer sitio para más caché de disco mientras que Linux estaba manteniendo las páginas sin utilizar y dejando menos memoria disponible para la caché y para páginas de procesos. No sé si esto sigue siendo cierto a día de hoy.

5. Optimizaciones de Prefallo y de Rellenado con Ceros

Realizar un fallo de Memoria Virtual no es costoso y la página subyacente ya está cargada y simplemente puede ser mapeada en el proceso, pero puede ser costoso si hay muchas de ellas de forma regular. Un buen ejemplo de esto es ejecutar un programa como ls(1) o ps(1) una y otra vez. Si el programa binario está mapeado en la memoria pero no lo está en la tabla de páginas, entonces todas las páginas que serán accedidas por el programa generarán un fallo cada vez que el programa se ejecute. Esto es innecesario cuando las páginas en cuestión ya están en la Caché de Memoria Virtual, de modo que FreeBSD intentará pre-poblar las tablas de páginas de un proceso con aquellas páginas que ya están en la Caché de Memoria Virtual. Algo que FreeBSD no hace todavía es un pre-copy-on-write de ciertas páginas al hacer exec. Por ejemplo, si ejecutas el programa ls(1) mientras ejecutas `vmstat 1` notarás que siempre produce un cierto número de fallos de página, incluso cuando lo ejecutas una y otra vez. Estos son fallos de página de rellenados de ceros, no fallos de código de programa (que ya han sido pre-fallados). Realizar una pre-copia de páginas en un exec o fork es un área en el que ser sujeto de más estudio.

Un gran porcentaje de los fallos de página que se producen son fallos de rellenado de ceros. Habitualmente puedes verlo observando la salida del comando `vmstat -s`. Esto ocurre cuando un proceso accede a páginas de su área de BSS. Se espera que el área de BSS esté inicializada a cero pero el sistema de Memoria Virtual no se molesta en asignar ninguna memoria en absoluto hasta el momento en el que el proceso accede de verdad. Cuando se produce un fallo el sistema de Memoria Virtual no solo debe asignar una nueva página, tiene que inicializarla a cero también. Para optimizar la operación de rellenado de ceros el sistema de Memoria Virtual tiene la capacidad de pre-inicializar páginas a cero y marcarlas como tal, y solicitar páginas pre-inicializadas a cero

cuando ocurre un fallo de rellenado de ceros. La pre-inicialización a cero ocurren cuando la CPU está ociosa pero el número de páginas que el sistema pre-inicializa a cero está limitado para evitar destrozar las cachés de memoria. Este es un ejemplo excelente de cómo añadir complejidad al sistema de Memoria Virtual para optimizar el camino crítico.

6. Optimizaciones de la Tabla de Páginas

Las optimizaciones de la tabla de páginas constituyen la parte más controvertida del diseño de la Memoria Virtual de FreeBSD y ha mostrado cierta tensión con la llegada de uso serio de `mmap()`. Creo que esto en realidad es una característica de la mayor parte de los BSDS aunque no estoy seguro de cuándo se introdujo por primera vez. Hay dos optimizaciones principales. La primar es que las tablas de páginas hardware no contienen un estado persistente sino que pueden descartarse en cualquier momento con solo un pequeño sobre coste en la gestión. La segunda es que cada entrada en la tabla de páginas activas en el sistema tiene una estructura `pv_entry` que lo gobierna la cual está enlazada a la estructura `vm_page`. FreeBSD puede simplemente iterar sobre esos mapeos que se sabe que existen mientras Linux tiene que comprobar todas las tablas de páginas que *podrían* contener un mapeo específico para ver si es así, lo que puede provocar un sobre coste de O(n^2) en algunas situaciones. Por esto FreeBSD tiene a tomar mejores decisiones sobre qué páginas reutilizar o intercambiar cuando la memoria está bajo estrés, resultando en un mejor rendimiento bajo carga. Sin embargo, FreeBSD requiere ajustes del núcleo para acomodar situaciones con grandes espacios de direcciones compartidos como los que pueden darse en sistemas nuevos porque podría agotar las estructuras `pv_entry`.

Tanto Linux como FreeBSD necesitan trabajar en este área. FreeBSD trata de maximizar la ventaja de un modelo de mapeo activo potencialmente disperso (no todos los procesos necesitan mapear todas las páginas de una biblioteca compartida por ejemplo), mientras que Linux trata de simplificar sus algoritmos. FreeBSD en general tiene la venta del rendimiento a costa de gastar algo más de memoria extra, pero FreeBSD se desmorona en el caso donde un fichero grande está compartido de forma masiva entre cientos de procesos. Linux, por otro lado, se desmorona en el caso donde muchos procesos mapean pocas porciones de la misma biblioteca compartida y también se ejecuta de forma no-óptima cuando intenta determinar si una página puede ser reutilizada o no.

7. Coloreado de Páginas

Terminaremos con las optimizaciones de coloreado de páginas. El coloreado de páginas es una optimización de rendimiento diseñada para asegurar que el acceso a páginas contiguas en memoria virtual hacen el mejor uso posible de la caché del procesador. Hace mucho tiempo (es decir, más de 10 años) las cachés de los procesadores solían mapear memoria virtual en lugar de memoria física. Esto produjo un gran número de problemas que incluyen tener que limpiar la caché en cada cambio de contexto en algunos casos, y problemas con los alias de datos en la caché. De hecho, si no tienes cuidado, páginas contiguas en memoria virtual podrían terminar utilizando la misma página en la caché del procesador—llevando a desechar prematuramente datos cacheables y reduciendo el rendimiento de la CPU. Esto es cierto incluso en cachés asociativas multi direccionales (aunque el efecto se mitiga algo).

El código de asignación de memoria de FreeBSD implementa optimizaciones de coloreado de

páginas, lo que significa que el código se asignación de memoria intentará localizar páginas libres que son contiguas desde el punto de vista de la caché. Por ejemplo, si la página 16 de memoria física está asignada a la página 0 de la memoria virtual del proceso y la caché puede mantener 4 páginas, el código de coloreado de páginas no asignará la página 20 de memoria física a la página 1 de la memoria virtual de un proceso. En su lugar, asignaría la página 21 de memoria física. El código de coloreado de páginas intenta evitar la asignación de la página 20 porque esto mapea sobre la misma memoria cacheada que la página 16 y resultaría en un cacheo no óptimo. Este código añade una significativa complejidad al subsistema de asignación de memoria de la Memoria Virtual como puedes imaginar, pero el resultado merece la pena. El Coloreado de Páginas hace que la memoria de la Memoria Virtual sea tan determinista como la memoria física en términos de rendimiento de caché.

8. Conclusión

La Memoria Virtual en lo sistemas operativos modernos deben afrontar diversas situaciones de forma eficiente y para muchos patrones de uso distintos. La aproximación modular y algorítmica que históricamente ha tomado BSD nos permite estudiar y entender la implementación actual así como reemplazar piezas de código relativamente grandes de forma también relativamente limpia. Ha habido una serie de mejoras en el sistema e Memoria Virtual de FreeBSD en los últimos años, y el trabajo continua.

9. Sesión extra de Preguntas y Respuestas por Allen Briggs

9.1. ¿Qué es el algoritmo de entrelazado al que hiciste referencia en la lista de problemas del sistema de intercambio de FreeBSD 3.X?

FreeBSD utiliza un entrelazado de intercambio fijo con un valor por defecto de 4. Esto significa que FreeBSD reserva espacio para cuatro áreas de intercambio incluso si solo tienes una, dos o tres. Puesto que el espacio de intercambio está entrelazado el espacio lineal de direcciones que representa las "cuatro áreas de intercambio" estará fragmentado si en realidad no tienes cuatro áreas de intercambio. Por ejemplo, si tienes dos áreas de intercambio A y B la representación del espacio de direcciones en FreeBSD para ese área de intercambio estará entrelazada en bloques de 16 páginas:

```
A B C D A B C D A B C D A B C D
```

FreeBSD 3.X utiliza una aproximación de "lista secuencial de regiones libres" para contabilizar las áreas de intercambio libres. La idea es que grandes bloques de espacio lineal libre puede ser representado con un único nodo en la lista (kern/subr_rlist.c). Pero debido a la fragmentación la lista termina estando completamente fragmentada. En el ejemplo superior, espacio de intercambio completamente sin utilizar hará que A y B se muestren como "libre" y C y D como "todo asignado".

Cada secuencia A-B requiere un nodo en la lista para ser contabilizado porque C y D son huecos, así que el nodo de la lista no puede ser combinado junto con la siguiente secuencia A-B.

¿Por qué entrelazamos nuestro espacio de intercambio en lugar de mover las áreas hacia el final y hacer algo más interesante? Es mucho más fácil asignar rondas lineales de un espacio de direcciones y luego entrelazar automáticamente el resultado en múltiples discos en lugar de tratar de poner toda esa sofisticación en otro lado.

La fragmentación causa otros problemas. Al utilizar una lista lineal en 3.X, y tener una cantidad tan grande de fragmentación, asignar y liberar intercambio termina siendo un algoritmo O(N) en lugar de un algoritmo O(1). Junto con otros factores (mucho acceso al intercambio) y empiezas a tener niveles de sobrecarga de orden O(N^2) y O(N^3), lo que es malo. El sistema 3.X puede necesitar además asignar Memoria Virtual del Núcleo durante una operación de intercambio para crear un nuevo nodo en la lista lo que puede producir un bloqueo si el sistema está intentando desalojar páginas en una situación de memoria baja.

En 4.X no utilizamos una lista secuencial. En su lugar utilizamos un árbol radix y mapas de bits de bloques de intercambio en lugar de nodos de listas por rangos. Sufrimos la penalización de preasignar todos los mapas de bits necesarios para todo el área de intercambio pero esto al final desaprovecha menos memoria debido al uso de un mapa de bits (un bit por bloque) en lugar de una lista enlazada de nodos. El uso del árbol radix en lugar de una lista secuencia nos proporciona un rendimiento de casi O(1) independientemente de cómo de fragmentado esté el árbol.

9.2. ¿Cómo se relaciona la separación de páginas limpias y sucias (inactivas) con la situación donde puedes ver contadores bajos de la lista de cache y contadores altos de la lista activa en `systat -vm`? ¿Las estadísticas de systat cuentan las páginas activas y las sucias de forma conjunta en el contador de la cola activa?

Sí, eso es confuso. La relación es "objetivo" versus "realidad". Nuestro objeto es separar las páginas pero la realidad es que si no estamos en una crisis de memoria, en realidad no necesitamos hacerlo.

Esto significa que FreeBSD no intentará demasiado fuerte separar las páginas sucias (cola inactiva) de las limpias (cola de caché) cuando el sistema no está bajo estrés, ni intentará desactivar páginas (cola activa → cola inactiva) cuando el sistema no está bajo estrés, incluso si no están siendo utilizadas.

9.3. En el ejemplo de ls(1) / `vmstat 1`, algunos de los fallos de página no serían fallos de páginas de datos (COW del fichero del ejecutable a una página privada)? Es decir, esperaría algunos fallos de página fueran de rellenado de ceros y otros de datos de programa. ¿O te refieres a que FreeBSD hace pre-COW para los datos de programa?

Un fallo COW puede ser de rellenado de ceros o de datos de programa. El mecanismo es el mismo en cualquier caso porque el los datos de respaldo del programa ya estarán en la caché. De hecho estoy mezclando los dos. FreeBSD no hace pre-COW de los datos de programa o de rellenado de ceros, pero *sí* premapea páginas que existen en la caché.

9.4. En la sección de optimizaciones de la tabla de páginas, puedes dar algo más de detalle acerca de `pv_entry` y `vm_page` (o debería vm_page ser `vm_pmap`—como en 4.4, cf. pp. 180-181 de McKusick, Bostic, Karel, Quarterman)? Específicamente, ¿qué tipo de operación/reacción requeriría un escaneo de los mapas?

Un `vm_page` representa una tupla (objeto,índice#). Un `pv_entry` representa una entrada de la tabla de páginas hardware (pte). Si tienes cinco procesos compartiendo la misma página física y la tabla de páginas de tres de esos procesos mapean la página, ésta será representada mediante una sola estructura `vm_page` y tres estructuras `pv_entry`.

Las estructuras `pv_entry` sólo representan páginas mapeadas por la MMU (una `pv_entry` representa una pte). Esto significa que cuando necesitamos eliminar todas las referencias hardware a la `vm_page` (para reutilizar la página para otra cosa, pasarla a disco, borrarla, marcarla como sucia y demás) podemos simplemente escanear la lista enlazada de estructuras `pv_entry` asociadas con esa `vm_page` y eliminar o modificar la pte de sus tablas de páginas.

En Linux no existe dicha lista enlazada. Para eliminar todos los mapeos de tablas de páginas hardware para una `vm_page` linux debe acceder a cada objeto de Memoria Virtual que *podría* haber mapeado la página. Por ejemplo, si tienes 50 procesos todos mapeando la misma biblioteca compartida y quieres eliminar la página X de esa biblioteca, necesitas acceder a la tabla de páginas de cada uno de esos 50 procesos incluso si sólo 10 de ellos han mapeado la página. Así que Linux está favoreciendo la simplicidad en el diseño por el rendimiento. Muchos algoritmos de Memoria Virtual que son O(1) o (una N pequeña) en FreeBSD terminan siendo O(N), O(N^2), o peor en Linux. Puesto que los pte que representan una página concreta en un objeto suelen estar en el mismo desplazamiento en todas las tablas de páginas en las que están mapeadas, reducir el número de

accesos a las tablas de páginas en el mismo desplazamiento del pte evitará por lo general que se destruya la línea de caché L1 para ese desplazamiento, lo que puede conllevar un mejor rendimiento.

FreeBSD tiene más complejidad (el esquema de `pv_entry`) para mejorar el rendimiento (para limitar los accesos a la tabla de páginas *sólo* a aquellos pte que necesitan ser modificados).

Pero FreeBSD tiene un problema de escalado que Linux no tiene en cuento a que hay un número limitado de estructuras `pv_entry` y esto causa problemas cuando tienes datos masivamente compartidos. En esta caso podrías agotar las estructuras `pv_entry` incluso si hay memoria libre disponible de sobra. Esto se puede solucionar bastante fácilmente aumentando el número de estructuras `pv_entry` en la configuración del núcleo, pero necesitamos encontrar una forma mejor de hacerlo.

Respecto a la sobrecarga de memoria de una tabla de páginas versus el esquema de `pv_entry`: Linux utiliza tablas de páginas "permanentes" que no se descartan, pero no necesita una `pv_entry` para cada pte potencialmente mapeado. FreeBSD utiliza tablas de páginas "desechables" pero añade una estructura `pv_entry` para cada pte que esté realmente mapeado. Creo que la utilización de memoria termina siendo la misma, dándole a FreeBSD una ventaja algorítmica con su habilidad para desechar tablas de páginas a voluntad con muy poca sobrecarga.

9.5. Por último, en la sección de coloreado de páginas, podría ayudar describir un poco más a lo que te refieres. No lo seguí del todo.

¿Sabes cómo funciona una memoria caché hardware L1? Lo explicaré: Imagina una máquina con 16MB de memoria principal pero sólo 128K de caché L1. Normalmente esta caché funciona de modo que cada bloque de 128K de memoria principal utiliza *los mismos* 128K de caché. Si accedes al desplazamiento 0 en memoria principal y luego al desplazamiento 128L en memoria principal ¡terminas descartando los datos cacheados que leíste del desplazamiento 0!

Ahora bien, esto simplificando mucho las cosas. Lo que he descrito es lo que se llama una caché de memoria hardware de "mapeo directo". La mayoría de cachés modernas son lo que se llaman cachés asociativas de conjuntos de doble sentido o cachés asociativas de conjuntos de cuádruple sentido. La asociación por conjuntos te permite acceder hasta N regiones de memoria distintas que se solapan en la misma memoria de caché sin destruir los datos cacheados previamente. Pero sólo N.

Así que si tenemos una caché de conjuntos asociativa de cuádruple sentido puedo acceder los desplazamientos 0, 128K, 256K y 384K y todavía ser capaz de acceder al desplazamiento 0 de nuevo y que me lo devuelva de la caché L1. Se luego accedo al desplazamiento 512K, sin embargo, uno de loas cuatro objetos de datos cacheados previamente será descartado por la caché.

Es extremadamente importante… *extremadamente* importante que la mayoría de accesos a memoria del procesador vengan de la caché L1, porque la caché L1 opera a la frecuencia del procesador. En el momento en el que tienes una pérdida en la caché L1 y tienes que ir a la caché L2 o a la memoria principal, el procesador parará y potencialmente se sentaría a esperar durante un tiempo equivalente a *cientos* de instrucciones hasta que la lectura de memoria principal se

complete. La memoria principal (la memoria dinámica que pones en tu ordenador) es *lenta*, cuando se compara con la velocidad del procesador.

Ok, ahora vamos con el coloreado de páginas: Todas las memorias caché modernas con lo que se conoce como cachés *físicas*. Cachean direcciones de memoria física, no direcciones de memoria virtual. Esto permite no molestar a la caché durante un cambio de contexto de procesos, lo que es muy importante.

Pero en el mundo UNIX® tú tratas con espacios de direcciones virtuales, no espacios de direcciones físicas. Cualquier programa que escribas verá un espacio de direcciones virtuales que se le ha proporcionado. Las páginas virtuales *reales* que están por debajo del espacio de direcciones virtuales ¡no están necesariamente contiguas físicamente! De hecho, podrías tener dos páginas que están pegadas una a la otra en el espacio de direcciones del proceso y que terminan estando en el desplazamiento 0 y el desplazamiento 128K en memoria *física*.

Un programa normalmente asume que dos páginas que están una al lado de la otra serán cacheadas de forma óptima. Es decir, que puedes acceder a objetos de datos en ambas páginas sin tener que destrozar las entradas de caché de la otra página. Pero esto sólo es cierto si las páginas físicas bajo el espacio de memoria virtual son contiguas (en lo que a la caché se refiere).

Esto es lo que hace el coloreado de páginas. En lugar de asignar páginas físicas de forma *aleatoria*, lo que podría resultar en un rendimiento de caché no óptimo, el coloreado de Páginas asigna páginas físicas *razonablemente contiguas* a direcciones virtuales. Por lo tanto los programas se pueden escribir asumiendo que las características de la caché hardware subyacente son las mismas para el espacio de direcciones virtuales a como serían si el programa estuviera ejecutándose directamente en un espacio de direcciones físicas.

Nótese que digo "razonablemente" contiguas en lugar de simplemente "contiguas". Desde el punto de vista de una caché de mapeo directo de 128K, la dirección física 0 es la misma que la dirección física 128K. De modo que dos páginas una al lado de la otra en tu espacio de memoria virtual podrían terminar siendo el desplazamiento 128K y 132K en memoria física, pero podría fácilmente ser también el desplazamiento 128K y 4K en memoria física y mantener todavía las mismas características de rendimiento de la caché. Así que el coloreado de páginas *no* tiene que asignar páginas de memoria física realmente contiguas a páginas de memoria virtual que sí lo son, sólo necesita asegurarse de que asigna páginas contiguas desde el punto de vista del rendimiento y la operativa de la caché.