Elementi di progettazione del sistema di VM di FreeBSD

Matthew Dillon

dillon@apollo.backplane.com

$FreeBSD: release/8.4.0/it_IT.ISO8859-15/articles/vm-design/article.xml 39632 2012-10-01 11:56:00Z gabor $

$FreeBSD: release/8.4.0/it_IT.ISO8859-15/articles/vm-design/article.xml 39632 2012-10-01 11:56:00Z gabor $

Questo articolo è stato pubblicato in origine nel numero di gennaio 2000 di DaemonNews (http://www.daemonnews.org/). Questa versione dell'articolo può includere aggiornamenti da parte di Matt e di altri autori per riflettere i cambiamenti nell'implementazione della VM di FreeBSD.

Il titolo è in realtà solo un modo complicato per dire che cercherò di descrivere l'intera enchilada della memoria virtuale (VM), sperando di farlo in una maniera che chiunque possa seguire. Nell'ultimo anno mi sono concentrato su un certo numero di sottosistemi principali del kernel in FreeBSD, trovando quelli della VM (la memoria virtuale) e dello Swap i più interessanti, e considerando quello di NFS "un lavoretto necessario". Ho riscritto solo piccole porzioni di quel codice. Nell'arena della VM la sola grossa riscrittura che ho affrontato è stata quella del sottosistema di swap. La maggior parte del mio lavoro è stato di pulizia e mantenimento, con solo alcune moderate riscritture di codice e nessuna correzione rilevante a livello algoritmico nel sottosistema della VM. Il nocciolo della base teorica del sottosistema rimane immutato ed un bel po' del merito per gli sforzi di modernizzazione negli ultimi anni appartiene a John Dyson e David Greenman. Poiché non sono uno storico come Kirk non tenterò di marcare tutte le varie caratteristiche con i nomi delle relative persone, perché sbaglierei invariabilmente.

Traduzione a cura di Gabriele Renzi <`surrender_it@yahoo.it`>.

Sommario

1 Introduzione

Prima di andare avanti con la descrizione del progetto effettivo della VM spendiamo un po' di tempo sulla necessità di mantenere e modernizzare una qualunque base di codice longeva. Nel mondo della programmazione, gli algoritmi tendono ad essere più importanti del codice ed è dovuto alle radici accademiche di BSD che si è prestata grande attenzione alla progettazione algoritmica sin dal principio. Una maggiore attenzione al design in genere conduce ad una base di codice flessibile e pulita che può essere modificata abbastanza semplicemente, estesa, o rimpiazzata nel tempo. Mentre BSD viene considerato un sistema operativo "vecchio" da alcune persone, quelli di noi che lavorano su di esso tendono a considerarlo come una base di codice "matura" che ha vari componenti modificati, estesi, o rimpiazzati con codice moderno. Questa si è evoluta, e FreeBSD è all'avanguardia, non importa quanto possa essere vecchio qualche pezzo di codice. Questa è una distinzione importante da fare ed una di quelle che sfortunatamente sfuggono alla maggior parte delle persone. Il più grande errore che un programmatore possa fare è non imparare dalla storia, e questo è precisamente l'errore che molti sistemi operativi moderni hanno commesso. Windows NT® è il miglior esempio di questo, e le conseguenze sono state disastrose. Anche Linux commette questo errore a un certo livello—abbastanza perché noi appassionati di BSD possiamo scherzarci su ogni tanto, comunque. Il problema di Linux è semplicemente la mancanza di esperienza e di una storia con la quale confrontare le idee, un problema che sta venendo affrontato rapidamente dalla comunità Linux nello stesso modo in cui è stato affrontato da quella BSD—con il continuo sviluppo di codice. La gente di Windows NT, d'altro canto, fa ripetutamente gli stessi errori risolti da UNIX® decadi fa e poi impiega anni nel risolverli. E poi li rifanno, ancora, e ancora. Soffrono di un preoccupante caso di "non è stato progettato qui" e di "abbiamo sempre ragione perché il nostro dipartimento marketing dice così". Io ho pochissima tolleranza per chiunque non impari dalla storia.

La maggior parte dell'apparente complessità di progettazione di FreeBSD, specialmente nel sottosistema VM/Swap, è una conseguenza diretta dell'aver dovuto risolvere importanti problemi di prestazioni legati a varie condizioni. Questi problemi non sono dovuti a cattivi progetti algoritmici ma sorgono invece da fattori ambientali. In ogni paragone diretto tra piattaforme, questi problemi diventano più evidenti quando le risorse di sistema cominciano ad essere stressate. Mentre descrivo il sottosistema VM/Swap di FreeBSD il lettore dovrebbe sempre tenere a mente almeno due punti. Primo, l'aspetto più importante nel design prestazionale è ciò che è noto come "Ottimizzazione del Percorso Critico". Accade spesso che le ottimizzazioni prestazionali aggiungano un po di impurità al codice per far migliorare il percorso critico. Secondo, un progetto solido e generalizzato, funziona meglio di un progetto pesantemente ottimizzato, alla lunga. Mentre un progetto generale può alla fin fine essere più lento di un sistema pesantemente ottimizzato quando vengono implementati inizialmente, il progetto generalizzato tende ad essere più semplice da adattare alle condizioni variabili mentre quello pesantemente ottimizzato finisce per dover essere gettato via. Ogni base di codice che dovrà sopravvivere ed essere mantenibile per anni deve dunque essere progettata con

attenzione fin dall'inizio anche se questo può portare a piccoli peggioramenti nelle prestazioni. Vent'anni fa c'era ancora gente che sosteneva che programmare in assembly era meglio che programmare in linguaggi di alto livello, perché si poteva produrre codice che era dieci volte più veloce. Oggi, la fallacia di tale argomento è ovvia—così come i paralleli con il design algoritmico e la generalizzazione del codice.

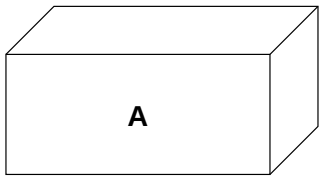
2 Oggetti VM

Il modo migliore per iniziare a descrivere il sistema di VM di FreeBSD è guardandolo dalla prospettiva di un processo a livello utente. Ogni processo utente vede uno spazio di indirizzamento della VM singolo, privato e contiguo, contenente molti tipi di oggetti di memoria. Questi oggetti hanno varie caratteristiche. Il codice del programma e i dati del programma sono effettivamente un singolo file mappato in memoria (il file binario che è stato eseguito), ma il codice di programma è di sola lettura mentre i dati del programma sono copy-on-write [1]. Il BSS del programma è solamente una zona di memoria allocata e riempita con degli zero su richiesta, detta in inglese "demand zero page fill". Nello spazio di indirizzamento possono essere mappati anche file arbitrari, che è in effetti il meccanismo con il quale funzionano le librerie condivise. Tali mappature possono richiedere modifiche per rimanere private rispetto al processo che le ha effettuate. La chiamata di sistema fork aggiunge una dimensione completamente nuova al problema della gestione della VM in cima alla complessità già data.

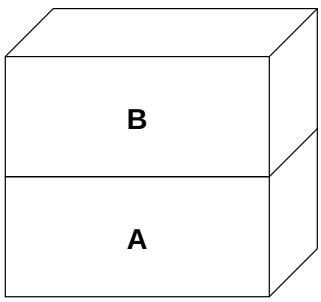
Una pagina di dati di un programma (che è una basilare pagina copy-on-write) illustra questa complessità. Un programma binario contiene una sezione di dati preinizializzati che viene inizialmente mappata direttamente in memoria dal file del programma. Quando un programma viene caricato nello spazio di memoria virtuale di un processo, questa area viene inizialmente copiata e mappata in memoria dal binario del programma stesso, permettendo al sistema della VM di liberare/riusare la pagina in seguito e poi ricaricarla dal binario. Nel momento in cui un processo modifica questi dati, comunque, il sistema della VM deve mantenere una copia privata della pagina per quel processo. Poiché la copia privata è stata modificata, il sistema della VM non può più liberarlo, poiché non ci sarebbe più nessuna possibilità di recuperarlo in seguito.

Noterai immediatamente che quella che in origine era soltanto una semplice mappatura di un file è diventata qualcosa di più complesso. I dati possono essere modificati pagina per pagina mentre una mappatura di file coinvolge molte pagine alla volta. La complessità aumenta ancora quando un processo esegue una fork. Quando un processo esegue una fork, il risultato sono due processi—ognuno con il proprio spazio di indirizzamento privato, inclusa ogni modifica fatta dal processo originale prima della chiamata a `fork()`. Sarebbe stupido per un sistema di VM creare una copia completa dei dati al momento della `fork()` perché è abbastanza probabile che almeno uno dei due processi avrà bisogno soltanto di leggere da una certa pagina da quel momento in poi, permettendo di continuare ad usare la pagina originale. Quella che era una pagina privata viene di nuovo resa una copy-on-write, poiché ogni processo (padre e figlio) si aspetta che i propri cambiamenti rimangano privati per loro e non abbiano effetti sugli altri.

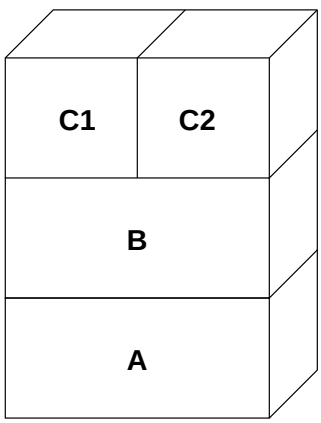
FreeBSD gestisce tutto ciò con un modello a strati di oggetti VM. Il file binario originale del programma risulta come lo strato di Oggetti VM più basso. Un livello copy-on-write viene messo sopra questo per mantenere quelle pagine che sono state copiate dal file originale. Se il programma modifica una pagina di dati appartenente al file originale il sistema dell VM prende un page fault [2] e fa una copia della pagina nel livello più alto. Quando un processo effettua una fork, vengono aggiunti altri livelli di Oggetti VM. Tutto questo potrebbe avere un po' più senso con un semplice esempio. Una `fork()` è un'operazione comune per ogni sistema *BSD, dunque questo esempio prenderà in considerazione un programma che viene avviato ed esegue una fork. Quando il processo viene avviato, il sistema della VM crea uno starto di oggetti, chiamiamolo A:



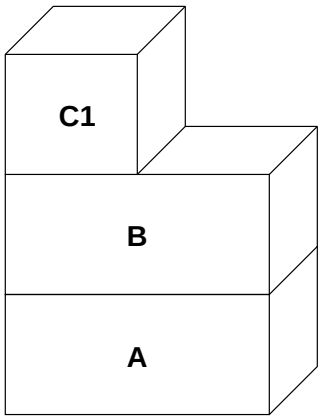
A rappresenta il file—le pagine possono essere spostate dentro e fuori dal mezzo fisico del file se necessario. Copiare il file dal disco è sensato per un programma, ma di certo non vogliamo effettuare il page out [3] e sovrascrivere l'eseguibile. Il sistema della VM crea dunque un secondo livello, B, che verrà copiato fisicamente dallo spazio di swap:



Dopo questo, nella prima scrittura verso una pagina, viene creata una nuova pagina in B, ed il suo contenuto viene inizializzato con i dati di A. Tutte le pagine in B possono essere spostate da e verso un dispositivo di swap. Quando il programma esegue la fork, il sistema della VM crea due nuovi livelli di oggetti—C1 per il padre e C2 per il figlio—che restano sopra a B:



In questo caso, supponiamo che una pagina in B venga modificata dal processo genitore. Il processo subirà un fault di copy-on-write e duplicherà la pagina in C1, lasciando la pagina originale in B intatta. Ora, supponiamo che la stessa pagina in B venga modificata dal processo figlio. Il processo subirà un fault di copy-on-write e duplicherà la pagina in C2. La pagina originale in B è ora completamente nascosta poiché sia C1 che C2 hanno una copia e B potrebbe teoricamente essere distrutta (se non rappresenta un "vero" file); comunque, questo tipo di ottimizzazione non è triviale da realizzare perché è di grana molto fine. FreeBSD non effettua questa ottimizzazione. Ora, supponiamo (come è spesso il caso) che il processo figlio effettui una `exec()`. Il suo attuale spazio di indirizzamento è in genere rimpiazzato da un nuovo spazio di indirizzamento rappresentante il nuovo file. In questo caso il livello C2 viene distrutto:



In questo caso, il numero di figli di B scende a uno, e tutti gli accessi a B avvengono attraverso C1. Ciò significa che B e C1 possono collassare insieme in un singolo strato. Ogni pagina in B che esista anche in C1 viene cancellata da B durante il crollo. Dunque, anche se l'ottimizzazione nel passo precedente non era stata effettuata, possiamo recuperare le pagine morte quando il processo esce o esegue una `exec()`.

Questo modello crea un bel po' di problemi potenziali. Il primo è che ci si potrebbe ritrovare con una pila abbastanza profonda di Oggetti VM incolonnati che costerebbe memoria e tempo per la ricerca quando accadesse un fault. Può verificarsi un ingrandimento della pila quando un processo esegue una fork dopo l'altra (che sia il padre o il figlio). Il secondo problema è che potremmo ritrovarci con pagine morte, inaccessibili nella profondità della pila degli Oggetti VM. Nel nostro ultimo esempio se sia il padre che il figlio modificano la stessa pagina, entrambi hanno una loro copia della pagina e la pagina originale in B non è più accessibile da nessuno. Quella pagina in B può essere liberata.

FreeBSD risolve il problema della profondità dei livelli con un'ottimizzazione speciale detta "All Shadowed Case" (caso dell'oscuramento totale). Questo caso accade se C1 o C2 subiscono sufficienti COW fault (COW è l'acronimo che sta per copy on write) da oscurare completamente tutte le pagine in B. Ponimo che C1 abbia raggiunto questo livello. C1 può ora scavalcare B del tutto, dunque invece di avere C1->B->A e C2->B->A adesso abbiamo C1->A e C2->B->A. ma si noti cos'altro è accaduto—ora B ha solo un riferimento (C2), dunque possiamo far collassare B e C2 insieme. Il risultato finale è che B viene cancellato interamente e abbiamo C1->A e C2->A. Spesso accade che B contenga un grosso numero di pagine e ne' C1 ne' C2 riescano a oscurarlo completamente. Se eseguiamo una nuova fork e creiamo un insieme di livelli D, comunque, è molto più probabile che uno dei livelli D sia eventualmente in grado di oscurare completamente l'insieme di dati più piccolo rappresentato da C1 o C2. La stessa ottimizzazione funzionerà in ogni punto nel grafico ed il risultato di ciò è che anche su una macchina con moltissime fork le pile degli Oggetti VM tendono a non superare una profondità di 4. Ciò è vero sia per il padre che per il figlio ed è vero nel caso sia il padre a eseguire la fork ma anche se è il figlio a eseguire fork in cascata.

Il problema della pagina morta esiste ancora nel caso C1 o C2 non oscurino completamente B. A causa delle altre ottimizzazioni questa eventualità non rappresenta un grosso problema e quindi permettiamo semplicemente alle pagine di essere morte. Se il sistema si trovasse con poca memoria le manderebbe in swap, consumando un po' di swap, ma così è.

Il vantaggio del modello ad Oggetti VM è che `fork()` è estremamente veloce, poiché non deve aver luogo nessuna copia di dati effettiva. Lo svantaggio è che è possibile costruire un meccanismo a livelli di Oggetti VM relativamente complesso che rallenterebbe la gestione dei page fault, e consumerebbe memoria gestendo le strutture degli Oggetti VM. Le ottimizzazioni realizzate da FreeBSD danno prova di ridurre i problemi abbastanza da poter essere ignorati, non lasciando nessuno svantaggio reale.

3 Livelli di SWAP

Le pagine di dati private sono inizialmente o pagine copy-on-write o pagine zero-fill. Quando avviene un cambiamento, e dunque una copia, l'oggetto di copia originale (in genere un file) non può più essere utilizzato per salvare la copia quando il sistema della VM ha bisogno di riutilizzarla per altri scopi. A questo punto entra in gioco lo SWAP. Lo SWAP viene allocato per creare spazio dove salvare memoria che altrimenti non sarebbe disponibile. FreeBSD alloca la struttura di gestione di un Oggetto VM solo quando è veramente necessario. Ad ogni modo, la struttura di gestione dello swap ha avuto storicamente dei problemi.

Su FreeBSD 3.X la gestione della struttura di swap prealloca un array che contiene l'intero oggetto che necessita di subire swap—anche se solo poche pagine di quell'oggetto sono effettivamente swappate questo crea una frammentazione della memoria del kernel quando vengono mappati oggetti grandi, o processi con grandi dimensioni all'esecuzione (large runsizes, RSS). Inoltre, per poter tenere traccia dello spazio di swap, viene mantenuta una "lista dei buchi" nella memoria del kernel, ed anche questa tende ad essere pesantemente frammentata. Poiché la "lista dei buchi" è una lista lineare, l'allocazione di swap e la liberazione hanno prestazioni non ottimali O(n) per ogni pagina. Questo richiede anche che avvengano allocazioni di memoria durante il processo di liberazione dello swap, e questo crea problemi di deadlock, blocchi senza uscita, dovuti a scarsa memoria. Il problema è ancor più esacerbato dai buchi creati a causa dell'algoritmo di interleaving. Inoltre il blocco di swap può divenire frammentato molto facilmente causando un'allocazione non contigua. Anche la memoria del Kernel deve essere allocata al volo per le strutture aggiuntive di gestione dello swap quando avviene uno swapout. È evidente che c'era molto spazio per dei miglioramenti.

Per FreeBSD 4.X, ho completamente riscritto il sottosistema di swap. Con questa riscrittura, le strutture di gestione dello swap vengono allocate attraverso una tabella di hash invece che con un array lineare fornendo una dimensione di allocazione fissata e una granularità molto maggiore. Invece di usare una lista lineare collegata per tenere traccia delle riserve di spazio di swap, essa usa una mappa di bit di blocchi di swap organizzata in una struttura ad albero radicato con riferimenti allo spazio libero nelle strutture nei nodi dell'albero. Ciò rende in effetti l'operazione di allocazione e liberazione delle risorse un'operazione O(1). L'intera mappa di bit dell'albero radicato viene anche preallocata in modo da evitare l'allocazione di memoria kernel durante le operazioni di swap critiche nei momenti in cui la memoria disponibile è ridotta. Dopo tutto, il sistema tende a fare uso dello swap quando ha poca memoria quindi dovremmo evitare di allocare memoria per il kernel in quei momenti per poter evitare potenziali deadlock. Infine, per ridurre la frammentazione l'albero radicato è in grado di allocare grandi spezzoni contigui in una volta, saltando i pezzetti frammentati. Non ho ancora compiuto il passo finale di avere un "puntatore di supportoall'allocazione" che scorra su una porzione di swap nel momento in cui vengano effettuate delle allocazioni, in modo da garantire ancor di più le allocazioni contigue o almeno una località nel riferimento, ma ho assicurato che un'aggiunta simile possa essere effettuata.

4 Quando liberare una pagina

Poiché il sistema della VM usa tutta la memoria disponibile per il caching del disco, in genere ci sono pochissime pagine veramente libere. Il sistema della VM dipende dalla possibilità di scegliere in maniera appropriata le pagine che non sono in uso per riusarle in nuove allocazioni. Selezionare le pagine ottimali da liberare è forse la funzione singola più importante che possa essere eseguita da una VM perché se si effettua una selezione non accurata, il sistema della VM può essere forzato a recuperare pagine dal disco in modo non necessari, degradando seriamente le prestazioni del sistema.

Quanto sovraccarico siamo disposti a sopportare nel percorso critico per evitare di liberare la pagina sbagliata? Ogni scelta sbagliata che facciamo ci costerà centinaia di migliaia di cicli di CPU ed uno stallo percettibile nei processi coinvolti, dunque permettiamo un sovraccarico significativo in modo da poter avere la certezza che la pagina scelta

sia quella giusta. Questo è il motivo per cui FreeBSD tende ad avere prestazioni migliori di altri sistemi quando le risorse di memoria vengono stressate.

L'algoritmo di determinazione della pagina da liberare è costruito su una storia di uso delle pagine di memoria. Per acquisire tale storia, il sistema si avvantaggia di una caratteristica della maggior parte dell'hardware moderno, il bit che indica l'attività di una pagina (page-used bit).

In qualsiasi caso, il page-used bit viene azzerato e in un momento seguente il sistema della VM passa di nuovo sulla pagina e vede che il page-used bit è stato di nuovo attivato. Questo indica che la pagina viene ancora usata attivamente. Il bit ancora disattivato è un indice che quella pagina non viene usata attivamente. Controllando questo bit periodicamente, viene sviluppata una storia d'uso (in forma di contatore) per la pagina fisica. Quando il sistema della VM avrà bisogno di liberare delle pagine, controllare questa storia diventa la pietra angolare nella determinazione del candidato migliore come pagina da riutilizzare.

E se l'hardware non ha un page-used bit?

Per quelle piattaforme che non hanno questa caratteristica, il sistema in effetti emula un page-used bit. Esso elimina la mappatura di una pagina, o la protegge, forzando un page fault se c'è un accesso successivo alla pagina. Quando avviene il page fault, il sistema segnala semplicemente la pagina come usata e la sprotegge in maniera che possa essere usata. Mentre prendere tale page fault solo per determinare se una pagina è in uso può apparire una scelta costosa, in realtà essa lo è molto meno che riusare la pagina per altri scopi, per dover poi scoprire che un processo ne aveva ancora bisogno e dovere andare a cercarla di nuovo su disco.

FreeBSD fa uso di parecchie code per le pagine per raffinare ulteriormente la selezione delle pagine da riutilizzare, come anche per determinare quando le pagine sporche devono essere spostate dalla memoria e immagazzinate da qualche parte. Poiché le tabelle delle pagine sono entità dinamiche in FreeBSD, non costa praticamente nulla eliminare la mappatura di una pagina dallo spazio di indirizzamento di un qualsiasi processo che la stia usando. Quando una pagina candidata è stata scelta sulla base del contatore d'uso, questo è esattamente quello che viene fatto. Il sistema deve effettuare una distinzione tra pagine pulite che possono essere teoricamente liberate in qualsiasi momento, e pagine sporche che devono prima essere scritte (salvate) per poter essere riutilizzabili. Quando una pagina candidata viene trovata viene spostata nella coda delle pagine inattive, se è una pagina sporca, o nella coda di cache se è pulita. Un algoritmo separato basato su un rapporto sporche/pulite determina quando le pagine sporche nella coda inattiva devono essere scritte su disco. Una volta che è stato fatto questo, le pagine ormai salvate vengono spostate dalla coda delle inattive alla coda di cache. A questo punto, le pagine nella coda di cache possono ancora essere riattivate da un VM fault ad un costo relativamente basso. Ad ogni modo, le pagine nella coda di cache vengono considerate "immediatamente liberabili" e verranno riutilizzate con un metodo LRU (least-recently used [4]) quando il sistema avrà bisogno di allocare nuova memoria.

È importante notare che il sistema della VM di FreeBSD tenta di separare pagine pulite e sporche per l'espressa ragione di evitare scritture non necessarie di pagine sporche (che divorano banda di I/O), e non sposta le pagine tra le varie code gratuitamente quando il sottosistema non viene stressato. Questo è il motivo per cui dando un `systat -vm` vedrai sistemi con contatori della coda di cache bassi e contatori della coda delle pagine attive molto alti. Quando il sistema della VM diviene maggiormente stressato, esso fa un grande sforzo per mantenere le varie code delle pagine ai livelli determinati come più efficenti. Per anni è circolata la leggenda urbana che Linux facesse un lavoro migliore di FreeBSD nell'evitare gli swapout, ma in pratica questo non è vero. Quello che stava effettivamente accadendo era che FreeBSD stava salvando le pagine inutilizzate proattivamente per fare spazio mentre Linux stava mantendendo le pagine inutilizzate lasciando meno memoria disponibile per la cache e le pagine dei processi. Non so se questo sia vero ancora oggi.

5 Pre-Faulting e Ottimizzazioni di Azzeramento

Subire un VM fault non è costoso se la pagina sottostante è già nella memoria fisica e deve solo essere mappata di nuovo nel processo, ma può divenire costoso nel caso se ne subiscano un bel po' su base regolare. Un buon esempio di ciò si ha eseguendo un programma come ls(1) o ps(1) ripetutamente. Se il binario del programma è mappato in memoria ma non nella tabella delle pagine, allora tutte le pagine che verranno accedute dal programmma dovranno generare un page fault ogni volta che il programma viene eseguito. Ciò non è necessario quando le pagine in questione sono già nella cache della VM, quindi FreeBSD tenterà di pre-popolare le tabelle delle pagine di un processo con quelle pagine che sono già nella VM Cache. Una cosa che FreeBSD non fa ancora è effettuare il pre-copy-on-write di alcune pagine nel caso di una chiamata a exec. Ad esempio, se esegui il programma ls(1) mentre stai eseguendo `vmstat 1` noterai che subisce sempre un certo numero di page fault, anche eseguendolo ancora e ancora. Questi sono zero-fill fault, legati alla necessità di azzerare memoria, non program code fault, legati alla copia dell'eseguibile in memoria (che erano già stati gestiti come pre-fault). Pre-copiare le pagine all'exec o alla fork è un'area che potrebbe essere soggetta a maggior studio.

Una larga percentuale dei page fault che accadono è composta di zero-fill fault. In genere è possibile notare questo fatto osservando l'output di `vmstat -s`. Questi accadono quando un processo accede a pagine nell'area del BSS. Ci si aspetta che l'area del BSS sia composta inizialmente da zeri ma il sistema della VM non si preoccupa di allocare nessuna memoria finché il processo non ne ha effettivamente bisogno. Quindi nel momento in cui accade un fault il sistema della VM non deve solo allocare una nuova pagina, ma deve anche azzerarla. Per ottimizzare l'operazione di azzeramento, il sistema della VM ha la capacità di pre-azzerare le pagine e segnalarle come tali, e di richiedere pagine pre-azzerate quando avvengono zero-fill fault. Il pre-azzeramento avviene quando la CPU è inutilizzata ma il numero di pagine che vengono pre-azzerate dal sistema è limitato per evitare di spazzare via la cache della memoria. Questo è un eccellente esempio di complessità aggiunta al sistema della VM per ottimizare il percorso critico.

6 Ottimizzazioni della Tabella delle Pagine

Le ottimizzazioni alla tabella delle pagine costituiscono La parte più controversa nel design della VM di FreeBSD ed ha mostrato un po' di affanno con l'avvento di un uso pesante di `mmap()`. Penso che questa sia una caratteristiche della maggior parte dei BSD anche se non sono sicuro di quando è stata introdotta la prima volta. Ci sono due ottimizzazioni maggiori. La prima è che le tabelle della pagine hardware non contengono uno stato persistente ma possono essere gettate via in qualsiasi momento con un sovraccarico di gestione minimo. La seconda è che ogni pagina attiva nel sistema ha una struttura di controllo `pv_entry` che è integrata con la struttura `vm_page`. FreeBSD può semplicemente operare attraverso quelle mappature di cui è certa l'esistenza, mentre Linux deve controllare tutte le tabelle delle pagine che *potrebbero* contenere una mappatura specifica per vedere se lo stanno effettivamente facendo, il che può portare ad un sovraccarico computazionale O(n^2) in alcune situazioni. È per questo che FreeBSD tende a fare scelte migliori su quale pagina riutilizzare o mandare in swap quando la memoria è messa sotto sforzo, fornendo una miglior performance sotto carico. Comunque, FreeBSD richiede una messa a punto del kernel per accomodare situazioni che richiedano grandi spazi di indirizzamento condivisi, come quelli che possono essere necessari in un sistema di news perché potrebbe esaurire il numero di struct `pv_entry`.

Sia Linux che FreeBSD necessitano di lavoro in quest'area. FreeBSD sta cercando di massimizzare il vantaggio di avere un modello di mappatura attiva potenzialmente poco denso (non tutti i processi hanno bisogno di mappare tutte le pagine di una libreria condivisa, ad esempio), mentre linux sta cercando di semplificare i suoi algoritmi. FreeBSD generalmente ha dei vantaggi prestazionali al costo di un piccolo spreco di memoria in più, ma FreeBSD crolla nel caso in cui un grosso file sia condiviso massivamente da centinaia di processi. Linux, d'altro canto, crolla nel caso in cui molti processi mappino a macchia di leopardo la stessa libreria condivisa e gira in maniera non ottimale anche quando cerca di determinare se una pagina deve essere riutilizzata o no.

7 Colorazione delle Pagine

Concluderemo con le ottimizzazioni di colorazione delle pagine. La colorazione delle pagine è un'ottimizzazione prestazionale progettata per assicurare che gli accessi a pagine contigue nella memoria virtuale facciano il miglior uso della cache del processore. Nei tempi antichi (cioè più di 10 anni fa) le cache dei processori tendevano a mapparela memoria virtuale invece della memoria fisica. Questo conduceva ad un numero enorme di problemi inclusa la necessità di ripulire la cache ad ogni cambio di contesto, in alcuni casi, e problemi con l'aliasing dei dati nella cache. Le cache dei processori moderni mappano la memoria fisica proprio per risolvere questi problemi. Questo significa che due pagine vicine nello spazio di indirizzamento dei processi possono non corrispondere a due pagine vicine nella cache. In effetti, se non si è attenti pagine affiancate nella memoria virtuale possono finire con l'occupare la stessa pagina nella cache del processore—portando all'eliminazione prematura di dati immagazzinabili in cache e riducendo le prestazioni della cache. Ciò è vero anche con cache set-associative [5] a molte vie (anche se l'effetto viene in qualche maniera mitigato).

Il codice di allocazione della memoria di FreeBSD implementa le ottimizizzazioni di colorazione delle pagine, ciò significa che il codice di allocazione della memoria cercherà di trovare delle pagine libere che siano vicine dal punto di vista della cache. Ad esempio, se la pagina 16 della memoria fisica è assegnata alla pagina 0 della memoria virtuale di un processo e la cache può contenere 4 pagine, il codice di colorazione delle pagine non assegnerà la pagina 20 di memoria fisica alla pagina 1 di quella virtuale. Invece, gli assegnerà la pagina 21 della memoria fisica. Il codice di colorazione delle pagine cerca di evitare l'assegnazione della pagina 20 perché questa verrebbe mappata sopra lo stesso blocco di memoria cache della pagina 16 e ciò causerrebbe un uso non ottimale della cache. Questo codice aggiunge una complessità significativa al sottosistema di allocazione memoria della VM, come si può ben immaginare, ma il gioco vale ben più della candela. La colorazione delle pagine rende la memoria virtuale deterministica quanto la memoria fisica per quel che riguarda le prestazioni della cache.

8 Conclusione

La memoria virtuale nei sistemi operativi moderni deve affrontare molti problemi differenti efficientemente e per molti diversi tipi di uso. L'approccio modulare ed algoritmico che BSD ha storicamente seguito ci permette di studiare e comprendere l'implementazione attuale cosi come di poter rimpiazzare in maniera relativamente pulita grosse sezioni di codice. Ci sono stati un gran numero di miglioramenti al sistema della VM di FreeBSD negli ultimi anni, ed il lavoro prosegue.

9 Sessione Bonus di Domande e Risposte di Allen Briggs

<briggs@ninthwonder.com>

1. Cos'è "l'algoritmo di interleaving" a cui fai riferimento nell'elenco delle debolezze della gestione dello swap in FreeBSD 3.X ?

FreeBSD usa un intervallo tra zone di swap fissato, con un valore predefinito di 4. Questo significa che FreeBSD riserva spazio per quattro aree di swap anche se ne hai una sola o due o tre. Poiché lo swap è intervallato lo spazio di indirizzamento lineare che rappresenta le "quattro aree di swap" verrà frammentato se non si possiedono veramente quattro aree di swap. Ad esempio, se hai due aree di swap A e B la rappresentazione dello spazio di FreeBSD per quell'area di swap verrà interrotta in blocchi di 16 pagine:

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

FreeBSD 3.X usa una "lista sequenziale delle regioni libere " per registrare le aree di swap libere. L'idea è che grandi blocchi di spazio libero e lineare possano essere rappresentati con un nodo singolo (`kern/subr_rlist.c`). Ma a causa della frammentazione la lista sequenziale risulta assurdamente frammentata. Nell'esempio precedente, uno spazio di swap completamente non allocato farà si che A e B siano mostrati come "liberi" e C e D come "totalmente allocati". Ogni sequenza A-B richiede un nodo per essere registrato perché C e D sono buchi, dunquei nodi di lista non possono essere combinati con la sequenza A-B seguente.

Perché organizziamo lo spazio in intervalli invece di appiccicare semplicemente le area di swap e facciamo qualcosa di più carino? Perché è molto più semplice allocare strisce lineari di uno spazio di indirizzamento ed ottenere il risultato già ripartito tra dischi multipli piuttosto che cercare di spostare questa complicazione altrove.

La frammentazione causa altri problemi. Essendoci una lista lineare nella serie 3.X, ed avendo una tale quantità di frammentazione implicita, l'allocazione e la liberazione dello swap finisce per essere un algoritmo O(N) invece di uno O(1). Combinalo con altri fattori (attività di swap pesante) e comincerai a trovarti con livelli di overhead come O(N^2) e O(N^3), e ciò è male. Il sistema dela serie 3.X può anche avere necessità di allocare KVM durante un'operazione di swap per creare un nuovo nodo lista, il che può portare ad un deadlock se il sistema sta cercando di liberare pagine nella memoria fisica in un momento di scarsità di memoria.

Nella serie 4.X non usiamo una lista sequenziale. Invece usiamo un albero radicato e mappe di bit di blocchi di swap piuttosto che nodi lista. Ci prendiamo il peso di preallocare tutte le mappe di bit richieste per l'intera area di swap ma ciò finisce per consumare meno memoria grazie all'uso di una mappa di bit (un bit per blocco) invece di una lista collegata di nodi. L'uso di un albero radicato invece di una lista sequenziale ci fornisce una performance quasi O(1) qualunque sia il livello di frammentazione dell'albero.

2. Non ho capito questo:

> È importante notare che il sistema della VM di FreeBSD tenta di separare pagine pulite e sporche per l'espressa ragione di evitare scritture non necessarie di pagine sporche (che divorano banda di I/O), e non sposta le pagine tra le varie code gratuitamente se il sottosistema non viene stressato. Questo è il motivo per cui dando un `systat -vm` vedrai sistemi con contatori della coda di cache bassi e contatori della coda delle pagine attive molto alti.

Come entra in relazione la separazione delle pagine pulite e sporche (inattive) con la situazione nella quale vediamo contatori bassi per la coda di cache e valori alti per la coda delle pagine attive in `systat -vm`? I dati di systat derivano da una fusione delle pagine attive e sporche per la coda delle pagine attive?

Si, questo può confondere. La relazione è "obiettivo" contro "realtà". Il nostro obiettivo è separare le pagine ma la realtà è che se non siamo in crisi di memoria, non abbiamo bisogno di farlo.

Questo significa che FreeBSD non cercherà troppo di separare le pagine sporche (coda inattiva) da quelle pulite (code della cache), ne cercherà di disattivare le pagine (coda pagine attive -> coda pagine inattive) quando il sistema non è sotto sforzo, anche se non vengono effettivamente usate.

3. Nell'esempio di ls(1) / `vmstat 1`, alcuni dei page fault non potrebbero essere data page faults (COW da file eseguibili a pagine private)? Cioè, io mi aspetterei che i page fault fossero degli zero-fill e dei dati di programma. O si implica che FreeBSD effettui il pre-COW per i dati di programma?

Un fault COW può essere o legato a uno zero-fill o a dati di programma. Il meccanismo è lo stesso in entrambi i casi poiché i dati di programma da copiare sono quasi certamente già presenti nella cache. E infatti li tratto insieme.

FreeBSD non effettua preventivamentela copia dei dati di programma o lo zero-fill, *effettua* la mappatura preventiva delle pagine che sono presenti nella sua cache.

4. Nella sezione sull'ottimizzazione della tabella delle pagine, potresti fornire maggiori dettagli su `pv_entry` e `vm_page` (forse vm_page dovrebbe essere `vm_pmap`—come in 4.4, cf. pp. 180-181 di McKusick, Bostic, Karel, Quarterman)? Specificamente, che tipo di operazioni/reazioni richiederebbero la scansione delle mappature?

Come funziona Linux nel caso in cui FreeBSD fallisce (la condivisione di un grosso file mappato tra molti processi)?

Una `vm_page` rappresenta una tupla (oggetto,indice#). Una `pv_entry` rappresenta una voce nella tabella delle pagine hardware (pte). Se hai cinque processi che condividono la stessa pagina fisica, e tre delle tabelle delle pagine di questi processi mappano effettivamente la pagina, questa pagina verrà rappresentata da una struttura `vm_page` singola e da tre strutture `pv_entry`.

Le strutture `pv_entry` rappresentano solo le pagine mappate dalla MMU (una `pv_entry` rappresenta un pte). Ciò significa che è necessario rimuovere tutti i riferimenti hardware a `vm_page` (in modo da poter riutilizzare la pagina per qualcos'altro, effettuare il page out, ripulirla, sporcarla, e così via) possiamo semplicemente scansionare la lista collegata di `pv_entry` associate con quella `vm_page` per rimuovere o modificare i pte dalla loro tabella delle pagine.

Sotto Linux non c'è una lista collegata del genere. Per poter rimuovere tutte le mappature della tabella delle pagine hardware per una `vm_page` linux deve indicizzare ogni oggetto VM che *potrebbe* aver mappato la pagina. Ad esempio, se si hanno 50 processi che mappano la stessa libreria condivisa e si vuole liberarsi della pagina X in quella libreria, sarà necessario cercare nella tabella delle pagine per ognuno dei 50 processi anche se solo 10 di essi ha effettivamente mappato la pagina. Così Linux sta barattando la semplicità del design con le prestazioni. Molti algoritmi per la VM che sono O(1) o (piccolo N) in FreeBSD finiscono per diventare O(N), O(N^2), o anche peggio in Linux. Poiché i pte che rappresentano una particolare pagina in un oggetto tendono ad essere allo stesso offset in tutte le tabelle delle pagine nelle quali sono mappati, la riduzione del numero di accessi alla tabela delle pagine allo stesso offset eviterà che la la linea di cache L1 per quell'offset venga cancellata, portando ad una performance migliore.

FreeBSD ha aggiunto complessità (lo schema `pv_entry`) in modo da incrementare le prestazioni (per limitare gli accessi alla tabella delle pagine *solo* a quelle pte che necessitino di essere modificate).

Ma FreeBSD ha un problema di scalabilità che linux non ha nell'avere un numero limitato di strutture `pv_entry` e questo provoca problemi quando si hanno condivisioni massicce di dati. In questo caso c'è la possibilità che finiscano le strutture `pv_entry` anche se c'è ancora una grande quantità di memoria disponibile. Questo può essere risolto abbastanza facilmente aumentando il numero di struttre `pv_entry` nella configurazione del kernel, ma c'è veramente bisogno di trovare un modo migliore di farlo.

Riguardo il sovrapprezzo in memoria di una tabella delle pagine rispetto allo schema delle `pv_entry`: Linux usa tabelle delle pagine "permanenti" che non vengono liberate, ma non necessita una `pv_entry` per ogni pte potenzialmente mappato. FreeBSD usa tabelle delle pagine "throw away", eliminabili, ma aggiunge una struttura `pv_entry` per ogni pte effettivamente mappato. Credo che l'utilizzo della memoria finisca per essere più o meno lo stesso, fornendo a FreeBSD un vantaggio algoritmico con la capacità di eliminare completamente le tabelle delle pagine con un sovraccarico prestazionale minimo.

5. Infine, nella sezione sulla colorazione delle pagine, potrebbe esser d'aiuto avere qualche descrizione in più di quello che intendi. Non sono riuscito a seguire molto bene.

Sai come funziona una memoria cache hardware L1? Spiego: Considera una macchina con 16MB di memoria

principale ma solo 128K di cache L1. In genere il modo in cui funziona la cache è che ogni blocco da 128K di memoria principale usa gli *stessi* 128K di cache. Se si accede all'offset 0 della memoria principale e poi al 128K si può finire per cancellare i dati che si erano messi nella cache dall'offset 0!

Ora, sto semplificando di molto. Ciò che ho appena descritto è quella che viene detta memoria cache a "corrispondenza diretta", o direct mapped. La maggior parte delle cache moderne sono quelle che vengono dette set-associative a 2 o 4 vie. L'associatività di questo tipo permette di accedere fino ad N regioni di memoria differenti che si sovrappongano sulla stessa cache senza distruggere i dati preventivamente immagazzinati. Ma solo N.

Dunque se ho una cache set associativa a 4 vie posso accedere agli offset 0, 128K, 256K 384K ed essere ancora in grado di accedere all'offset 0 ritrovandolo nella cache L1. Se poi accedessi all'offset 512K, ad ogni modo, uno degli oggetti dato immagazzinati precedentemente verrebbero cancellati dalla cache.

È estremamente importante … *estremamente* importante che la maggior parte degli accessi del processore alla memoria vengano dalla cache L1, poiché la cache L1 opera alla stessa frequenza del processore. Nel momento in cui si ha un miss [6] nella cache L1 si deveandare a cercare nella cache L2 o nella memoria principale, il processore andrà in stallo, e potenzialmente potrà sedersi a girarsi i pollici per un tempo equivalente a *centinaia* di istruzioni attendendo che la lettura dalla memoria principale venga completata. La memoria principale (la RAM che metti nel tuo computer) è *lenta*, se comparata alla velocità del nucleo di un moderno processore.

Ok, ora parliamo della colorazione dele pagine: tutte le moderne cache sono del tipo noto come cache *fisiche*. Esse memorizzano indirizzi di memoria fisica, non indirizzi di memoria virtual. Ciò permette alla cache di rimanere anche nel momento in cui ci sia un cambio di contesto tra processi, e ciò è molto importante.

Ma nel mondo UNIX devi lavorare con spazi di indirizzamento virtuali, non con spazi di indirizzamento fisici. Ogni programma che scrivi vedrà lo spazio di indirizzamento virtuale assegnatogli. Le effettive pagine *fisiche* nascoste sotto quello spazio di indirizzi virtuali non saranno necessariamente contigue fisicamente! In effetti, potresti avere due pagine affiancate nello spazio di indirizzamento del processo cge finiscono per trovarsi agli offset 0 e 128K nella memoria *fisica*.

Un programma normalmente assume che due pagine affiancate verranno poste in cache in maniera ottimale. Cioè, che possa accedere agli oggetti dato in entrambe le pagine senza che esse si cancellino a vicenda le rispettiva informazioni in cache. Ma ciò è vero solo se le pagine fisiche sottostanti lo spazio di indirizzo virtuale sono contigue (per quel che riguarda la cache).

Questo è ciò che viene fatto dalla colorazione delle pagine. Invece di assegnare pagine fisiche *casuali* agli indirizzi virtuali, che potrebbe causare prestazioni non ottimali della cache, la colorazione dele pagine assegna pagine fisiche *ragionevolmente contigue*. Dunque i programmi possono essere scritti assumendo che le caratteristiche per lo spazio di indirizzamento virtuale del programma della cache hardware sottostante siano uguali a come sarebbero state se avessero usato lo spazio di indirizzamento fisico.

Si note ho detto "ragionevolmente" contigue invece che semplicemente "contigue". Dal punto di vista di una cache di 128K a corrispondenza diretta, l'indirizzo fisico 0 è lo stesso che l'indirizzo fisico 128K. Dunque due agine affiancate nello spzio di indirizzamento virtuale potrebbero finire per essere all'offset 128K e al 132K nella memoria fisica, ma potrebbero trovarsi tranquillamente anche agli offset 128K e 4K della memoria fisica e mantenera comunque le stesse caratteristiche prestazionali nei riguardi della cache. Dunque la colorazione delle pagine *non* deveassegnare pagine di memoria fisica veramente contigue a pagine di memoria virtuale contigue, deve solo assicurarsi che siano assegnate pagine contigue dal punto di vista delle prestazioni/operazioni della cache.

Note

1. I dati copy on write sono dati che vengono copiati solo al momento della loro effettiva modifica
2. Un page fault, o “mancanza di pagina”, corrisponde ad una mancanza di una determinata pagina di memoria a un certo livello, ed alla necessità di copiarla da un livello più lento. Ad esempio se una pagina di memoria è stata spostata dalla memoria fisica allo spazio di swap su disco, e viene richiamata, si genera un page fault e la pagina viene di nuovo copiata in ram.
3. La copia dalla memoria al disco, l’opposto del page in, la mappatura in memoria.
4. Usate meno recentemente. Le pagine che non vengono usate da molto tempo probabilmente non saranno necessarie a breve, e possono essere liberate.
5. set-associative sta per associative all’interno di un insieme, in quanto c’è un insieme di blocchi della cache nei quale puo essere mappato un elemento della memoria fisica.
6. Un miss nella cache è equivalente a un page fault per la memoria fisica, ed allo stesso modo implica un accesso a dispositivi molto più lenti, da L1 a L2 come da RAM a disco.