Implementacja pamięci wirtualnej w systemach rodziny BSD

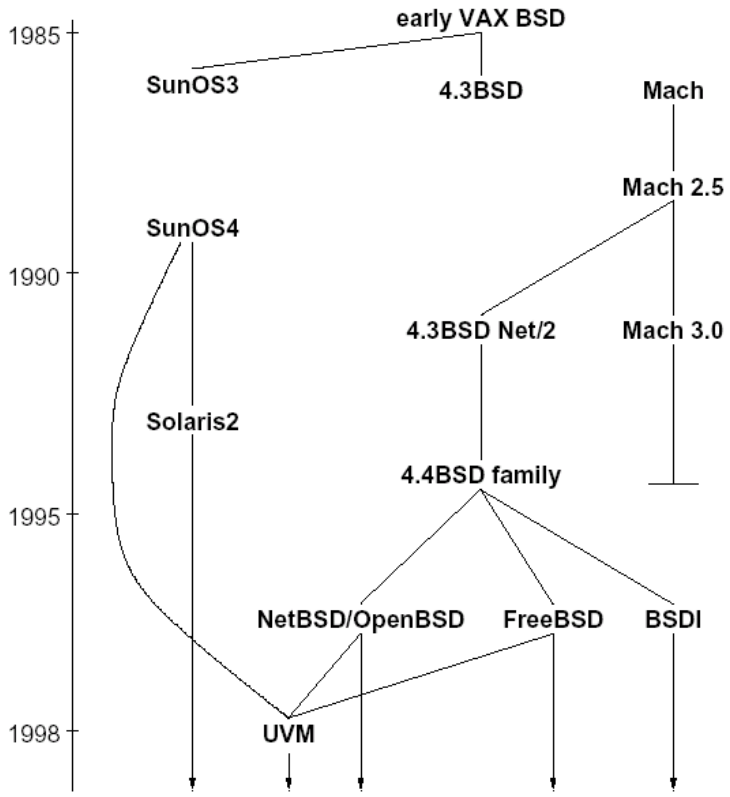
Łukasz Lew

13 grudnia 2002

Spis treści

1 Wstęp

1.1 Ewolucja VM i BSD VM



Rysunek 1: Ewolucja VM

VAX ograniczone możliwości procesora

Mach dobra separacja części zależnej i niezależnej od sprzętu

1.2 Dlaczego VM jest tak złożony?

1.2.1 Z czego VM korzysta

- MMU: translacja adresów, sygnalizacja błędu
- pamięć: d_mmap
- dysk(backing store): ffs
- sieć(backing store): nfs

1.2.2 Główna funkcjonalność VM

- pozwala na użycie jednolitej przestrzeni adresów
- alokacja pamięci fizycznej
- śledzenie zajętych i wolnych stron pamięci fizycznej
- alokacja dla każdego procesu virtulnej przestrzeni adresowej
- mapowanie fizycznych stron na virtualne (programowanie MMU)
- obsługa „page faults"
- przenoszenie danych pomiędzy „backing store", a pamięcią fizyczną (rządania do podsystemu I/O)
- zwalnianie nieużywanych stron
- przenoszenie do swap'a stron rzadko używanych
- duplikowanie przestrzeni adresowej podczas operacji fork

2 VM i cykl życia procesu (na przykładzie init)

2.1 Start procesu

1. tworzymy nową wirtualną przestrzeń adresową bez mapowań
2. mapujemy sekcje text i data z pliku /sbin/init
3. tworzymy wyzerowaną wirtualną przestrzeń bss oraz stack (fizyczne tworzenie dopiero przy próbie odczytu lub zapisu)
4. na obiektach mapujących text i data, ustawiamy atrubut copy_on_write
5. na obiekcie mapującym sekcję text ustawiamy read_only na innych read_write
6. text jest copy_on_write gdyż debugger może zmienić read_only na read_write i wstawiać breakpoint'y
7. data jest copy_on_write
8. bss i stack są r/w i bez copy_on_write - nie ma on dla niech znaczenia, gdyż jest to pamięć anonimowa, t.j. bez fizycznego obiektu (plik/urządzenie) i wyzerowana
9. przerwa pomiędzy bss i stack jest wykorzystywana dla otwieranych plików oraz bibliotek dzielonych (mmap)

2.2 Page Faults

1. próba wykonania pierwszej instrukcji spowoduje tylko odkrycie, że strony nie ma w pamięci
2. MMU generuje „page fault"
3. sprawdzamy czy to zapis czy odczyt oraz adres, który wygenerował błąd
4. wywołanie „VM page fault handler" z tymi danymi
5. szukamy w mapowaniach odpowiednich danych (np. pierwsza strona z /sbin/init)
6. jeżeli nie ma żadnych mapowań to VM generuje błąd „segmentation violation"
7. w.p.p wczytujemy dane z „backing store"
8. programujemy odpowiednio MMU

Kiedy ilość wolnych stron spadnie poniżej pewnego ustalonego poziomu wywoływany jest proces systemowy pagedeamon, który zwalnia nieużywane strony (dodaje je do free list), a jeżeli strona się zmieniła to najpierw zapisuje te dane do „backing store".

3 Architektura BSD VM

BSD VM na zewnątrz udostępnia pewien zestaw operacji. Natomiast wewnątrz BSD VM dzieli się na dwie główne części: Machine Dependent i Machine Independent (MD i MI). Taki podziałznacznie zwiększa przenośność gdyż MI jest dużo większą częścią systemu niż MD.

3.1 Operacje BSD VM

break zmienia rozmiar heap'u procesu

mmap/munmap mapuje/odmapowuje plik lub anonimową pamięć (wyzerowaną)

mprotect zmienia tryb ochrony mapowania: r/o, r/w, none

minherit zmienia atrybut inherit używany np przy fork'u. Może przyjmować stany copy/shared/none

msync upewnia, że strony których zmodyfikowane wersje są w pamięci zostaną zapisane do „backing store"

madvise doradza VM systemowi, co może zrobić z jakimś rejonem pamięci. Dopuszczalne parametry: normal, random, sequential, „will need", „won't need"

mlock zatrzymuje dane w pamięci fizycznej, nawet gdy są już niepotrzebne

swapctl pozwala na konfiguruję swap'u

3.2 Warstwa Machine Dependent

Każda architektura obsługiwana przez BSD VM ma swój moduł MD zwany pmap (physical map). Jest to po prostu interfejs służący do programowania MMU. Na obiekt pmap można patrzeć jak na dużą tablicę indeksowaną virtualnymi adresami stron, a zawierającą adresy fizyczne stron i ich atrybuty. Każdy proces w systemie używającym BSD VM ma swoją pmap'ę.

Dostępne operacje:

pmap_enter dodaje mapowanie do pmap'a z określonymi atrybutami ochrony (wywoływana w obsłudze page fault'a)

pmap_remove analogicznie

pmap_protect zmienia atrybuty ochrony przedziału pamięci we wskazanym pmap'ie (podczas copy_on_write)

pmap_page_protect zmienia atrybuty wskazanej strony we wszystkich pmap'ach związanych z tą stroną (wywoływana przed operacją pageout)

pmap_is_referenced/pmap_is_modified (używane przez pagedeamon'a gdy szuka stron do zwolnienia)

pmap_clear_reference/pmap_clear_modify używane przez pagedeamon'a, by mógł rozpoznać strony które należy zwolnić

pmap_copy używane w fork'u

3.3 Warstwa Machine Independent

MI jest wspólny dla wszystkich architektur, stanowi większą część systemu. Podstawowe struktury:

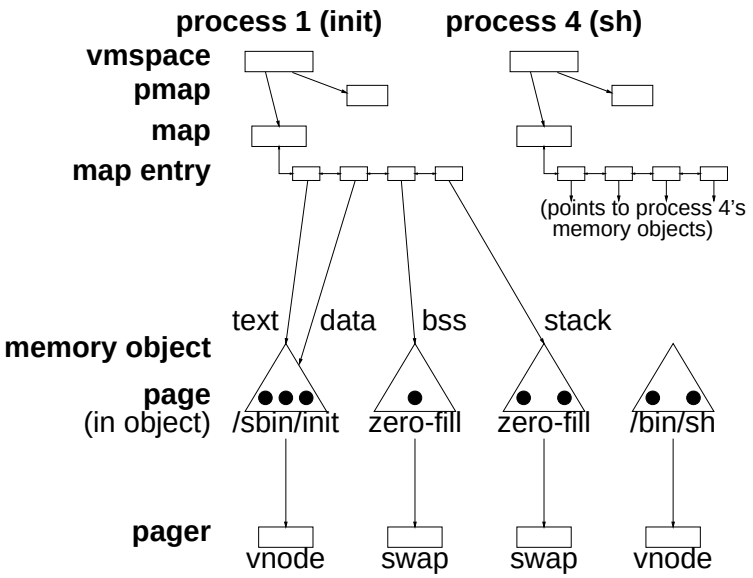
vmspace odpowiedzialny za pamięć wirtualną jeden z obiektów skojarzonych z procesem

vm_map opisuje wirtualną przestrzeń adresową prcesu lub jądra. Zawiera listę map_entry

vm_object opisuje plik, urządzenie lub anonimową pamięć (wyzerowaną),umożliwia ujednolicenie ich mapowania w wirualną pamięć

vm_pagers obiekt zawierający zestaw funkcji do obsługi „backing store"

vm_page opisuje stronę pamięci. Zazwyczaj dla każdej fizycznej strony istnieje dokładna struktura vm_page



Rysunek 2: Poglądowy rysunek na część MI BSD VM

Jak pokaznano na rysunku struktura vm_map odwzorowuje vm_object w przestrzeń adresową procesu. Dane należące do obiektów są przechowywane w strukturach vm_page. Dane z vm_page są wymieniane pomiędzy fizyczną pamięcią a „backing store" za pomocą vm_pagers. Każdy vm_map ma przypisane pmap.

Szczegóły implementacyjne:

vm_map zawiera posortowana po adresach dwukierunkowa lista map_entry

map_entry zwykle wskazuje na vm_object, ale istnieją dwa przypadki gdy wskazuje na inną mapę. Jeżeli kernel chce zmienić atrybut części pamięci opisywanej przez pojedyncze map_entry musi podzielić je na 2 lub 3 części.

sub-map'y używane tylko przez kernel, żeby podzielić przestrzeń adresową na mniejsze części (żeby uniknąć blokowania całości pamięci)

share-map pozwala procesom dzielić przestrzeń wirtualną.

vm_object Na jeden vm_object może wskazywać wiele map_entry'sów Obiekt zawiera listę stron, strony są identyfikowane na podstawie offset'u. Strony zazwyczaj są dodawane do vm_object w wyniku obsługi „page fault". Istnieje lista obiektów które mają licznik odwołań 0 - teoretycznie możliwych do zwolnienia w każdej chwili, ale używanych jako cache.

vm_pagers są trzy rodzaje:

- device

- swap
- vnode

vm_pages pageq lista wszystkich stron active/inactive/free

hashq globalny hash (vm_object, offset) => vm_page

listq lista stron, które należą do vm_object

4 Implementacja najciekawszych własności BSD VM

4.1 Atrybut copy_on_write

copy_on_write jak sama nazwa wskazuje jest to atrubut, który gdy ustawiony powoduje nie kopiowanie strony do momentu gdy nie wystąpiła zapis do niej. Wtedy by uniknąć zminy oryginału wykonywana jest kopia. Są dwa typy copy_on_write:

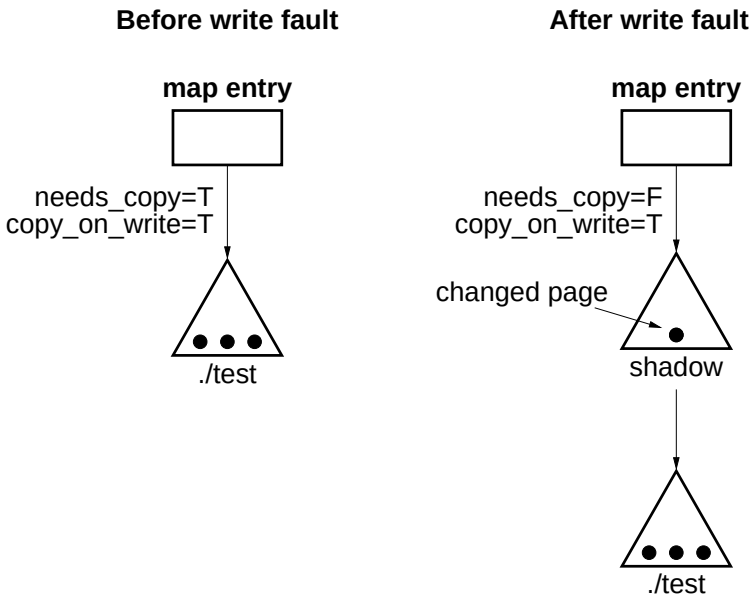
- private copy_on_write
- copy copy_on_write

W większości systemów Unix'owych copy copy_on_write nie jest obsługiwane ze względu na zbyt duży stopień komplikacji i zmniejszoną wydajność.

Obydwa używają „shadow objects" do trzymania zmodyfikowanych stron. Jest to dobry moment by przedstawić jak działa fork w BSD VM.

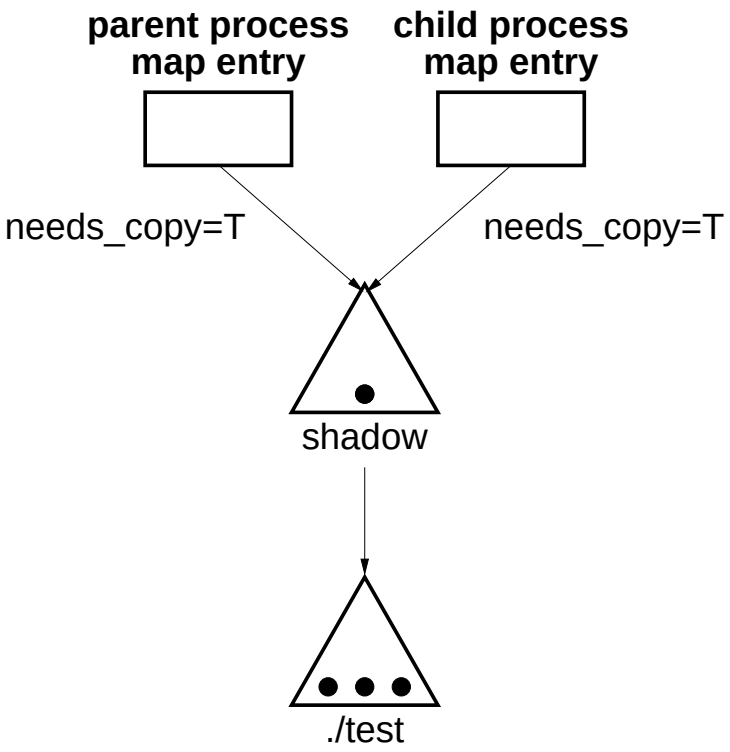
Atrybut needs_copy sygnalizuje czy obiekt jest prywatnym obiektem procesu tj. czy proces musi stworzyć kopię obiektu gdy chce do niego coś zapisać.

Jeden obraz to więcej niż 1000 słów, a więc:



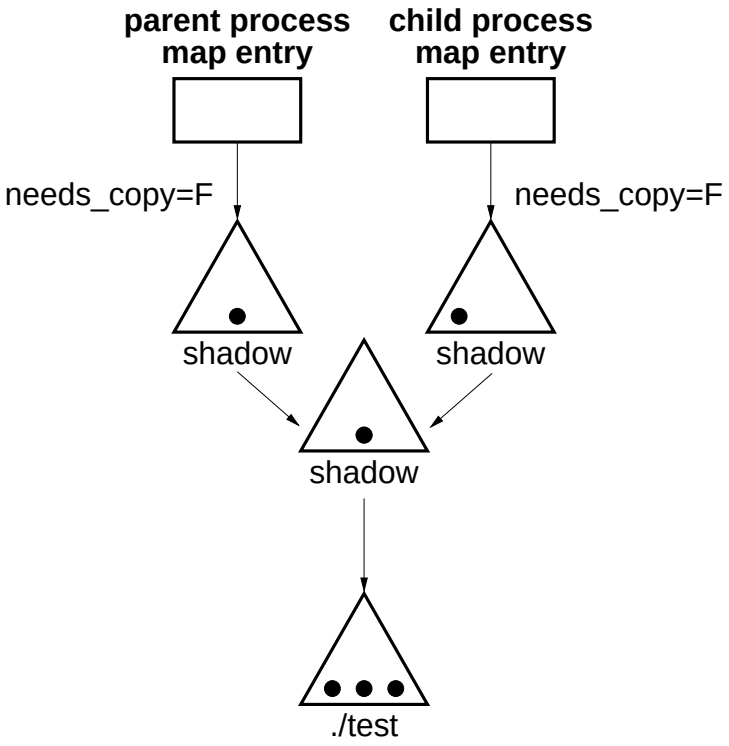
Rysunek 3: Mapowanie pliku z użyciem metody copy_on_write

W momencie gdy wystąpi write fault na pewnej stronie tworzony jest shadow object ze zmodyfikowaną stroną.



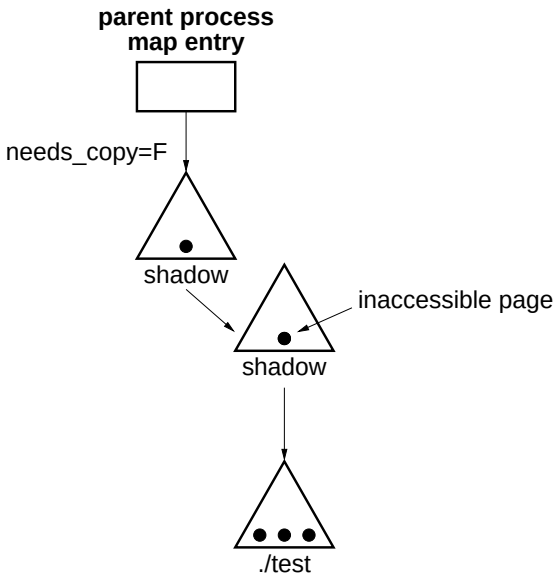
Rysunek 4: kontynuacja poprzednigo rysunku - sytuacja bezpośrednio po forku

Gdy proces się rozforkuje zarówno rodzic, jak i dziecko wskazują na ten sam shadow object, ale atrybut needs_copy powraca do stanu true.



Rysunek 5: oba procesy zapisały coś do swojej pamięci za (pomocą write page fault)

Teraz gdy procesy potomne wykonaja operacje zapisu tworzone są kolejne shadow object'y i atrybut needs_copy wraca do stanu false.

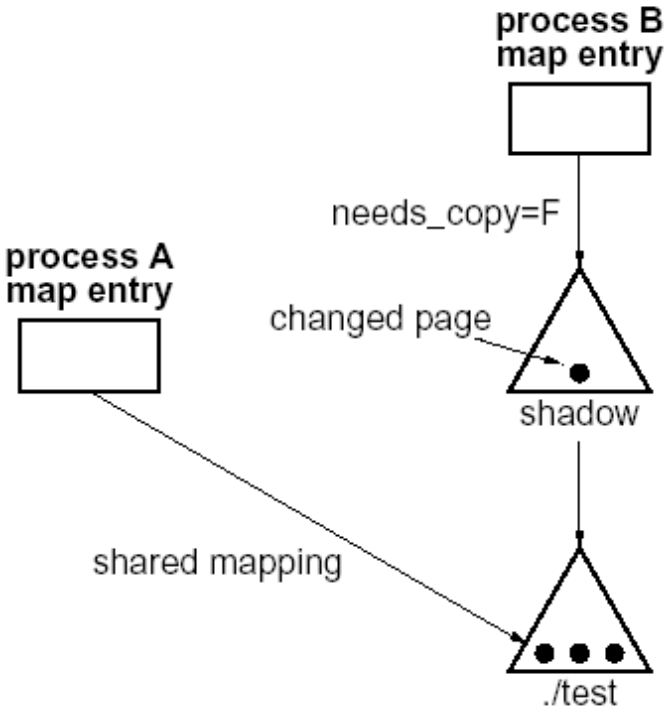


Rysunek 6: sytuacja gdy zostaje niepotrzebny vm_object - wyciek pamięci

I tu uwidacznia się jedna z największych wad systemu BSD VM. Jak widac na rysunku istnieje mozliwość istnienia w systemie niedostępnej strony, Jej usunięcie nie jest mozliwe. Takie strony moga się nagromadzić i zapchać swap, a następnie całą fizyczną pamięć, nasepuje zakleszczenie systemu.

Dystrybucje korzystające z BSD VM jakoś próbują radzić sobie z tym problemem, ale są to rozwiązania częściowe i bardzo złożone.

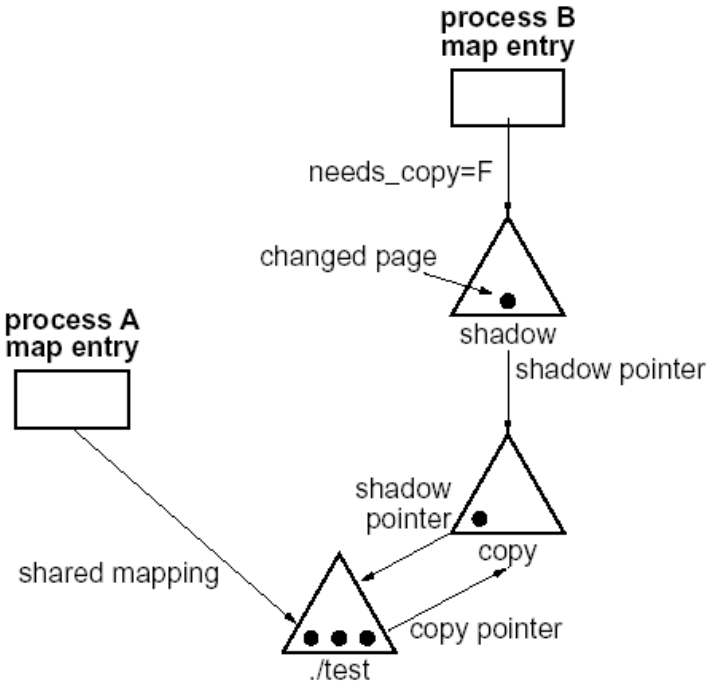
4.1.1 private copy_on_write



Rysunek 7: private copy_on_write - sytuacja gdy dwa procesy dostają się do jednego obiektu.

Jeżeli proces A dokona zmian przed B w jakiejś stronie, to B to zobaczy. Natomiast jeżeli proces A dokona zmian po B w jakiejś stronie, to B tego nie zobaczy.

4.1.2 copy copy_on_write



Rysunek 8: copy copy_on_write

W takim przypadku każdy z procesów będzie miał własną kopię obiektu.
Problemy:

- kolejne warunki, kolejne warianty wszystko to zwiększa złożoność BSD VM
- copy copy_on_write zmniejsza wydajności i zwiększa użycie pamięci
- problem wycieku pamięci staje się jeszcze bardziej skomplikowany

4.2 Algorytm page fault handler'a

1. szukamy w vm_map map_entry, który odpowiada adresowi, który spowodował page fault, jeżeli go nie znajdujemy otrzymujemy „segmentation violation"

2. w znalezionym vm_map_entry jest vm_object, w którym wyszukujemy vm_page. jeżeli nie znajdziemy to przechodzimy do kolejnych shadow'ów jak się skończą i nie znajdziemy odpowiedniej strony to błąd „segmentation fault"

3. jeżeli znajdziemy to jeżeli nie znajduje się ona w pamięci fizycznej, to ściągamy ją z „backing store"

4. zanim vm_pager zacznie operacje I/O celem pobrania strony, odblokowuje wszystkie struktury danych, a po ściągnięciu strony wszystko zostaje wykonane od początku.

5. następnie ustawiamy za pomocą pmap'a skojarzenie w MMU wraz z odpowiednimi atrybutami