Krótki przegląd systemu zarządzania pamięcią wirtualną w BSD

Michał Kozłowski

10 grudnia 2002

Spis treści

1 Wprowadzenie **3**

2 BSD 4.4VM **3**
2.1 Podział na warstwy . . . 3
2.1.1 Warstwa zależna od sprzętu . . . 3
2.2 Ogólny przegląd systemu zarządzania pamięcią - warstwa MI . . . 3
2.2.1 vm_space . . . 4
2.2.2 vm_map . . . 4
2.2.3 map_entry . . . 5
2.2.4 vm_object . . . 5
2.2.5 vm_page . . . 6
2.2.6 vm_pager . . . 6
2.3 Copy–on–write i obiekty przesłaniające . . . 7
2.4 Realizacja pamięci dzielonej . . . 7
2.5 Realizacja fork . . . 8
2.5.1 Problem długich łańcuchów i „wycieków” pamięci . . . 9

3 UVM **10**
3.1 Wstęp . . . 10
3.2 Co zostało z VM . . . 10
3.3 Co się zmieniło . . . 11
3.3.1 vm_anon . . . 12
3.3.2 vm_amap . . . 12
3.3.3 uvm_object . . . 13
3.4 Realizacja fork . . . 13
3.4.1 Przestrzeń adresowa współdzielona . . . 13
3.4.2 Przestrzeń adresowa ma być kopiowana . . . 14
3.4.3 Szczególne przypadki . . . 14
3.5 Podsumowanie . . . 15

4 Bibliografia **15**

1 Wprowadzenie

Ta prezentacja dotyczy podsystemów zarządzania pamięcią wirtualną w systemach z serii BSD. W pierwszej części omówiony zostanie „tradycyjny" moduł VM obecny w BSD 4.4. W dalszej części zostanie omówiony zastępujący go UVM, implementowany np. w darmowym NetBSD.

W obu przypadkach nacisk położony będzie na realizację *fork*. Uwypuklone zostaną różnice między prezentowanymi rozwiązaniami.

2 BSD 4.4VM

2.1 Podział na warstwy

Podsystem zarządzania pamięcią w BSD jest podzielony na dwie warstwy:

- Warstwa zależna od sprzętu (ang. machine dependent MD)
- Warstwa niezależna od sprzętu (ang. machine independent MI)

2.1.1 Warstwa zależna od sprzętu

Warstwa ta udostępnia abstrakcję fizycznej pamięci, wykorzystywaną przez warstwę niezależną od sprzętu. Trzonem warstwy MD jest struktura *pmap*. *pmap* przyporządkowuje wirtualne indeksy stron w fizyczne strony wraz z ich atrybutami (ochroną, bitami modyfikacji itp.). Można powiedzieć, że *pmap* jest w istocie jedną, wielką tablicą stron. Ponieważ każdy proces ma własną przestrzeń wirtualną, każdy ma swoją strukturę *pmap*

Do zarządzania strukturą *pmap* służą następujące funkcje:

pmap_enter tworzy nowe wiązania między wirtualnym indeksem, a fizyczną stroną ze wskazanymi atrybutami ochrony. Wywoływana w reakcji na *page fault* kiedy adresowana jest strona (wirtualna) nie związana z fizyczną

pmap_remove - usuwanie wiązania. Wywoływana przy zwalnianiu stron

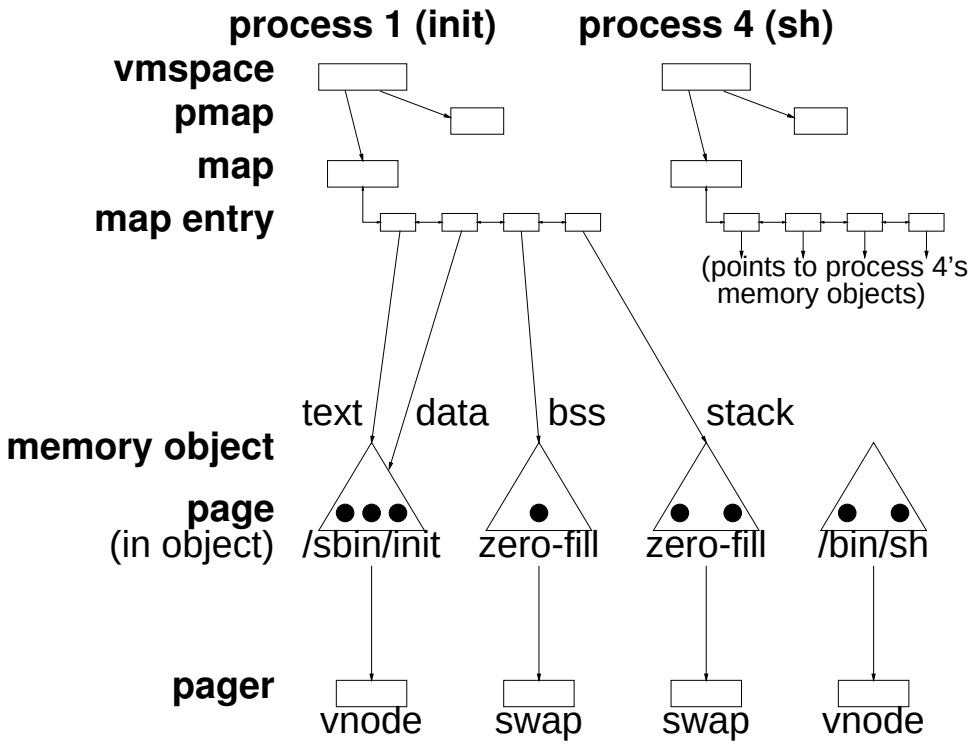
pmap_protect - ustawianie atrybutów ochrony strony

pmap_reference,pmap_modified - odczytywanie bitów dostępu/modyfikacji

pmap_clear_reference,pmap_celar_modify - zerowanie bitów dostępu/modyfikacji

2.2 Ogólny przegląd systemu zarządzania pamięcią - warstwa MI

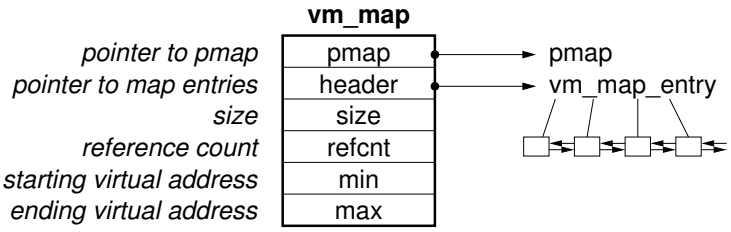
Poniższy rysunek przedstawia stan struktur danych warstwy MI bezpośrednio po załadowaniu dwóch procesów (tutaj init i sh).



2.2.1 vm_space

Zawiera opis wirtualnej przestrzeni wirtualnej. Składają się nań wskaźniki do *vm_map* i *pmap*

2.2.2 vm_map

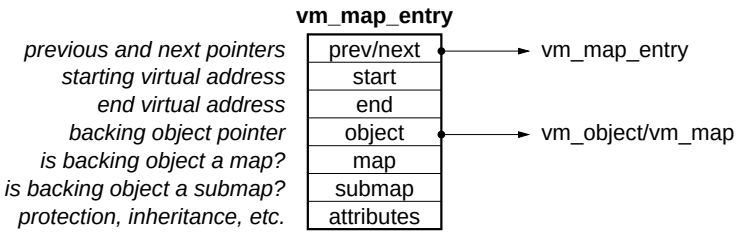


Zawiera opis przyporządkowania poszczególnych fragmentów wirtualnej przestrzeni adresowej do konkretnych obiektów. Składa się, z:

- wskaźnika do dwukierunkowej listy *map_entry*
- początku i końca mapowanej przestrzeni adresowej

– wskaźnika do *pmap* (nie pokazane na rysunku)

2.2.3 map_entry



map_entry opisuje przyporządkowanie konkretnego fragmentu wirtualnej przestrzeni adresowej do pewnego obiektu. *map_entry* są zorganizowane w dwukierunkową listę posortowaną względem adresu początku obszaru przez nie opisywanego. Obiektem przyporządkowanym do *map_entry* może być albo *vm_object* albo inna mapa (tzw. share map) - wykorzystywane jest to przy mechanizmie pamięci dzielonej (patrz niżej).

Istnieje cały zestaw funkcji do manipulowania *map_entry*. Pozwalają one np. na mapowanie nowych obszarów, usuwanie wiązań, zmianę atrybutów. Zauważmy, że czasami może to wymagać powstania nowych elementów *map_entry*, na przykład jeśli mamy przyporządkowanie adresów od 1000-5000 jako ReadOnly i chcemy teraz zmienić prawa dla adresów 3000-5000 na Read/Write, wówczas musi powstać nowy *map_entry* opisujący obszar 3000-5000, a stary zostanie ograniczony do obszaru 1000-3000.

Struktura *map_entry* zawiera następujące informacje:

– Wskaźniki do poprzedniego i następnego *map_entry*

– Początek i koniec mapowanego obszaru

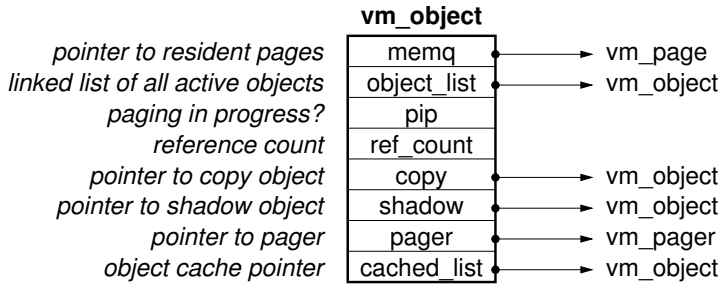
– Obiekt, na który mapowany jest obszar

– Znacznik czy obiekt jest *vm_object* czy mapą

– Atrybuty obszaru (ochrona, inne parametry np. copy–on–write patrz „Realizacja fork”)

2.2.4 vm_object

Pola *map_entry* wskazują na obiekty, które „okupują” wirtualną przestrzeń adresową. Do opisu tych obiektów służą struktury *vm_object*. Na obiekt składa się pewna (dowolna) liczba stron (opisywanych przez strukturę *vm_page* - patrz niżej. Dodatkowo w obiekcie znajdują się:



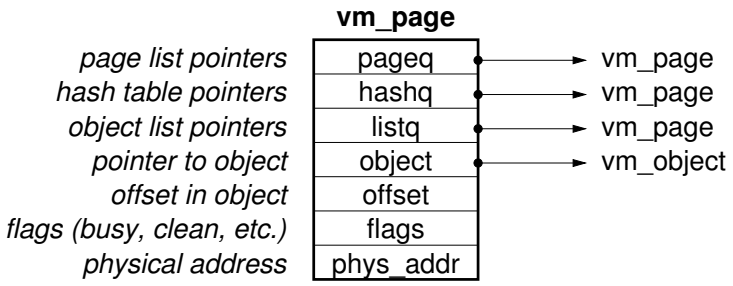
– licznik odwołań do niego

– obiekt, który posłuży do ładowania stron w przypadku ich braku w obiekcie, lub zapisywania jeśli zajdzie potrzeba wyrzucenia strony

– wskaźniki do obiektów kopiowania i przesłaniania (patrz „copy–on–write”)

Warto zwrócić uwagę na istnienia cache’a obiektów. W cache’u znajdują się obiekty, których licznik odwołań wynosi 0. Intencja przechowywania tych obiektów jest taka, żeby zachować obiekty, które są często używane. Np. w cache’u będzie znajdował się obiekt reprezentujący plik `/bin/ls`, pozwoli to zaoszczędzić wczytywania go za każdym razem z dysku. Sam obiekt zaś będzie używany krótko (bo `ls` wykonuje się szybko)

2.2.5 vm_page



Struktura *vm_page* reprezentuje pojedynczą stronę pamięci fizycznej. Struktura ta zawiera odwołanie do obiektu, do którego należy, adres fizyczny oraz atrybuty (ochrona, bity modyfikacji itd.). Kiedy strona jest wysyłana do przestrzeni wymiany znika obiekt *vm_page* z nią związany (przywracany jest przy próbie dostępu do strony o właściwym adresie w wyniku realizacji page-fault)

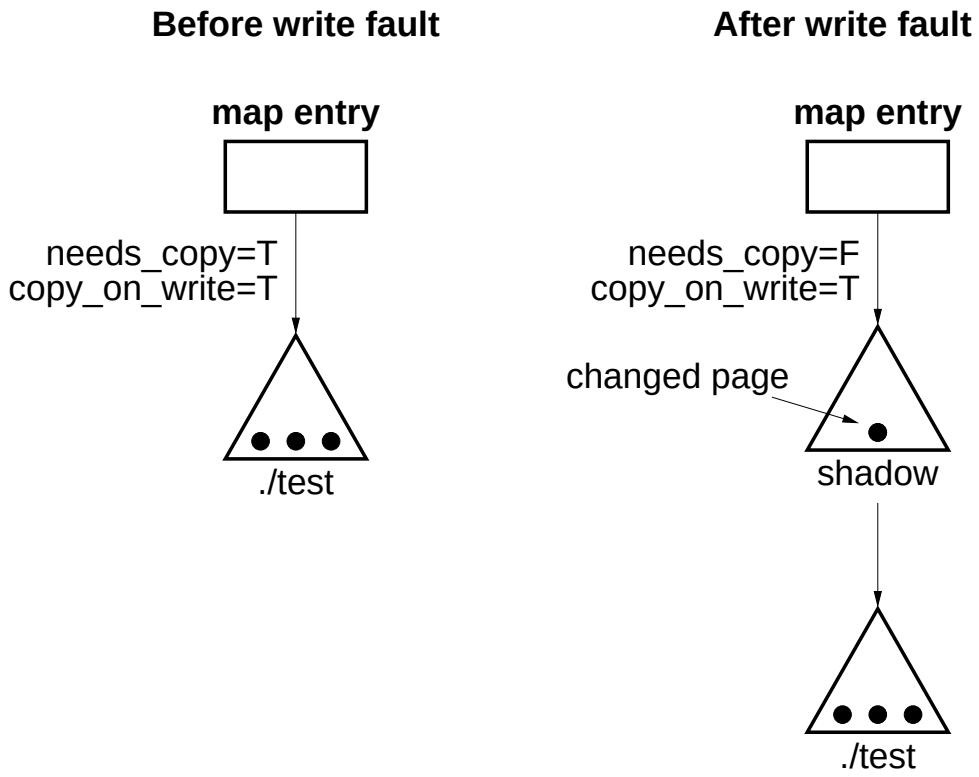
2.2.6 vm_pager

Reprezentuje obiekt, na którym dokonuje się wymiany stron. Może to być np. *vnode* reprezentujący plik albo *swap* reprezentujący przestrzeń wymiany.

2.3 Copy–on–write i obiekty przesłaniające

Załóżmy, że w sytuacji jak na rysunku powyżej proces init zechce coś zapisać do jednej ze stron w obiekcie reprezentującym „data”. Zauważmy, że nie może tego zrobić „bezpośrednio”, bo zmiany zostałyby ostatecznie zapisane z powrotem do pliku! Dlatego w *map_entry* opisującym region „data” procesu mamy ustawione atrybuty *needs–copy* oraz *copy–on–write*. Needs–copy oznacza, że przy próbie zapisu należy stworzy nowy obiekt, nazywany *obiektem przesłaniającym*. A copy–on–write, że w przypadku zapisu do strony, której nie ma w obiekcie przesłaniającym - należy ją skopiować do tego obiektu.

Ważne jest to, że w obiekcie przesłaniającym znajdują się tylko te strony, które zostały zmodyfikowane (bo chcemy uniknąć kopiowania). Żeby zapewnić dostęp do wszystkich stron obiekt przesłaniający i oryginalny muszą być połączone w łańcuch. Służy do tego pole shadow, które w obiekcie przesłaniającym ustawia się na obiekt oryginalny. Kiedy jest odwołanie do strony zarządza pamięci najpierw szuka jej w obiekcie przesłaniającym, następnie w oryginalnym. Kiedy tam go nie znajdzie następuje page fault.



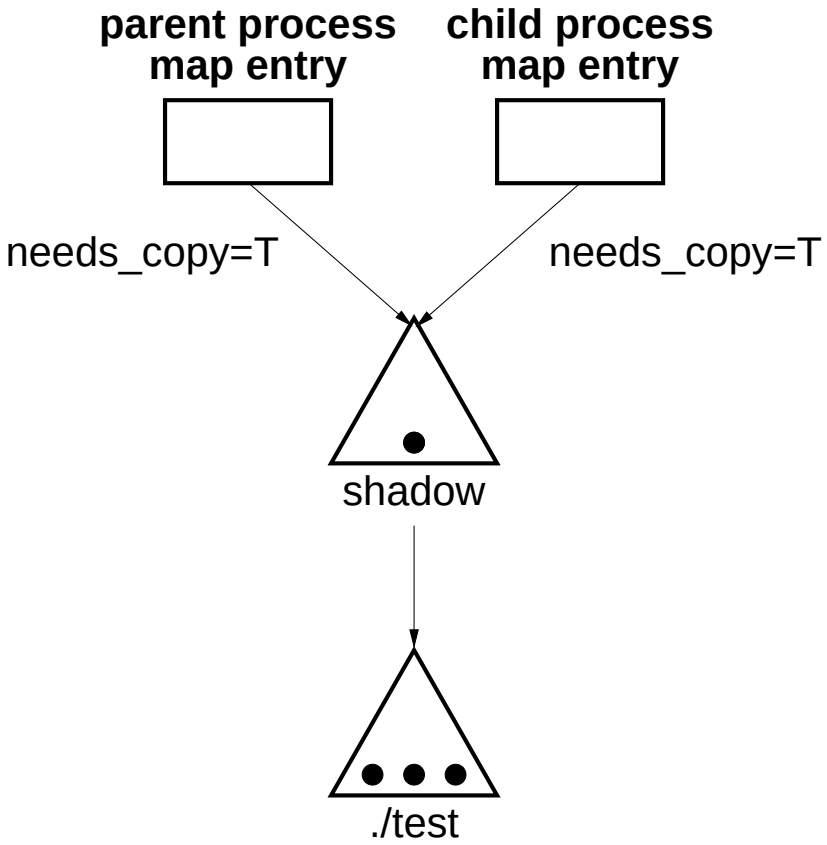
2.4 Realizacja pamięci dzielonej

Jeśli chcemy, żeby kilka procesów miało dostęp do tego samego obszaru pamięci *map_entry* zamiast obiekt musi mapować w następną mapę *vm_map*. Mapa ta będzie opisywać przestrzeń

dzieloną. Wszystkie procesy, które zawierają *map_entry* wskazujące na tę mapę będą miały dostęp do tej pamięci i będą widziały zmiany dokonywane przez pozostałe procesy

2.5 Realizacja fork

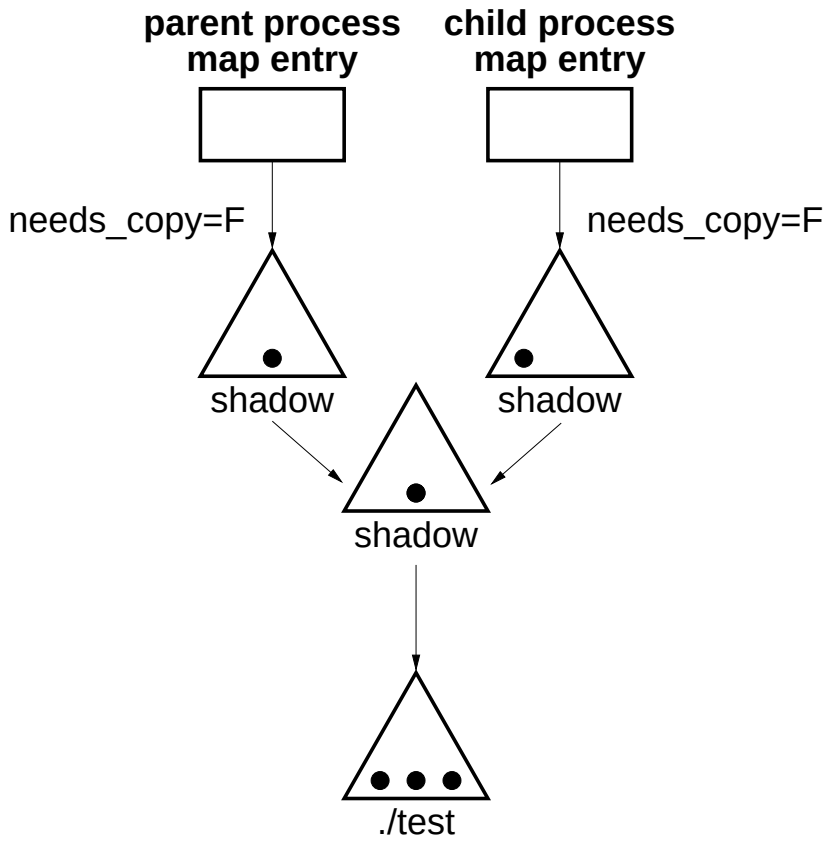
Mechanizm *copy–on–write* jest intensywnie wykorzystywany, przy realizacji funkcji systemowej *fork*. Bezpośrednio po wywołaniu *fork* sytuacja wygląda jak na rysunku poniżej:



Zauważmy, że po samym wywołaniu *fork* nie dokonuje się żadne kopiowanie danych. Przestrzeń adresowa obu procesów (macierzystego i potomnego) jest mapowana w ten sam obiekt. taka sytuacja będzie się utrzymywać do czasu próby pierwszego zapisu (ponieważ mapowanie jest z atrybutem „needs–copy” - patrz rysunek.

Kiedy, któryś z procesów spróbuje zapisu - powstaje obiekt przesłaniający, do którego kopiowana jest modyfikowana strona. Na rysunku poniżej przedstawiona jest sytuacja, w której proces macierzysty modyfikuje „środkową” stronę, a potomny - „lewą”.

Zauważmy, że flaga „needs–copy” została wyzerowana oraz, że łańcuch obiektów wydłużył się o jeden.

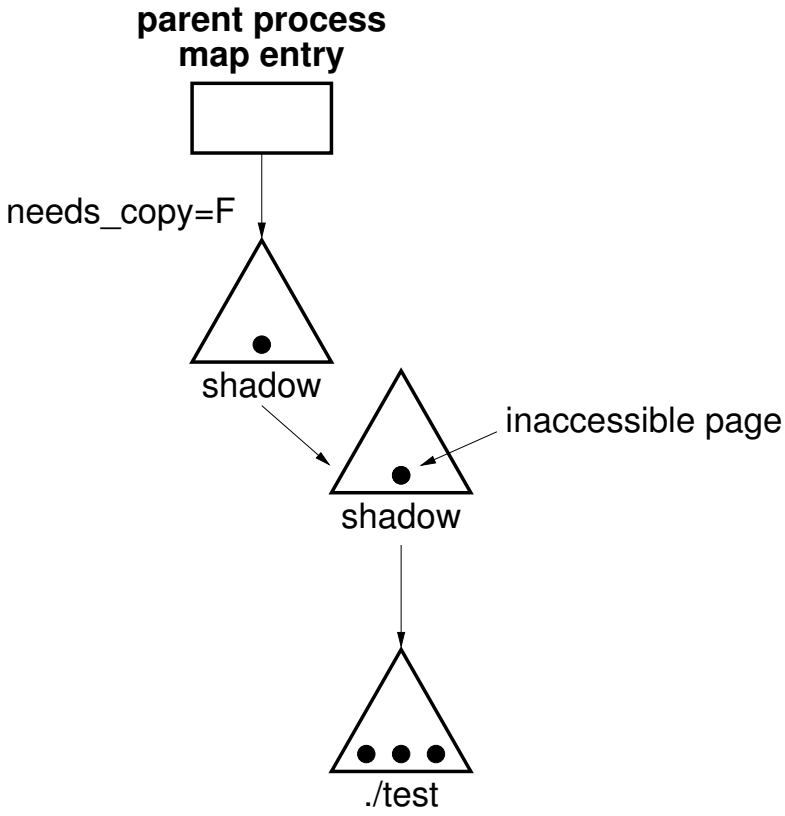


2.5.1 Problem długich łańcuchów i „wycieków” pamięci

Gdyby proces potomny również wykonał *fork* łańcuch obiektów wydłużyłby się jeszcze bardziej. W ramach postępowania tego procesu dostęp do pojedynczej strony będzie się wydłużał, ponieważ w niektórych przypadkach będzie to wymagała przejścia całego, potencjalnie długiego łańcucha obiektów.

Jednak taka implementacja *fork* powoduje także inny, znacznie poważniejszy problem. Wróćmy do sytuacji z poprzedniego rysunku i załóżmy, że proces potomny się kończy. Powstanie wówczas sytuacja jak na rysunku poniżej:

Strona ze środkowego obiektu przesłaniającego jest już niepotrzebna (w rzeczywistości cały ten obiekt jest niepotrzebny i należałoby go usunąć). Zarządca pamięci w BSD V4.4 nie rozwiązuje tego problemu. Niepotrzebna strona zostanie prędzej czy później wyrzucona do przestrzeni wymiany, ale nigdy nie będzie zwolniona (chyba, że zakończą się wszystkie procesy odwołujące się do tego łańcucha obiektów). W końcu może to doprowadzić do wyczerpania przestrzeni wymiany i w efekcie blokady sytemu. Problem ten nazywa się z angielskiego „swap memory leak”



3 UVM

3.1 Wstęp

UVM to nowy system pamięci wirtualnej, który zastępuje stary, stosowany w wersji 4.4 BSD. UVM przez uproszczenie struktur danych (np. wyeliminowanie łańcuchów obiektów), poprawia wydajność systemu. Przede wszystkim ulepszony został mechanizm copy–on–write przez całkowite jego przeprojektowanie, celem wyeliminowania problemów, o których mowa wyżej.

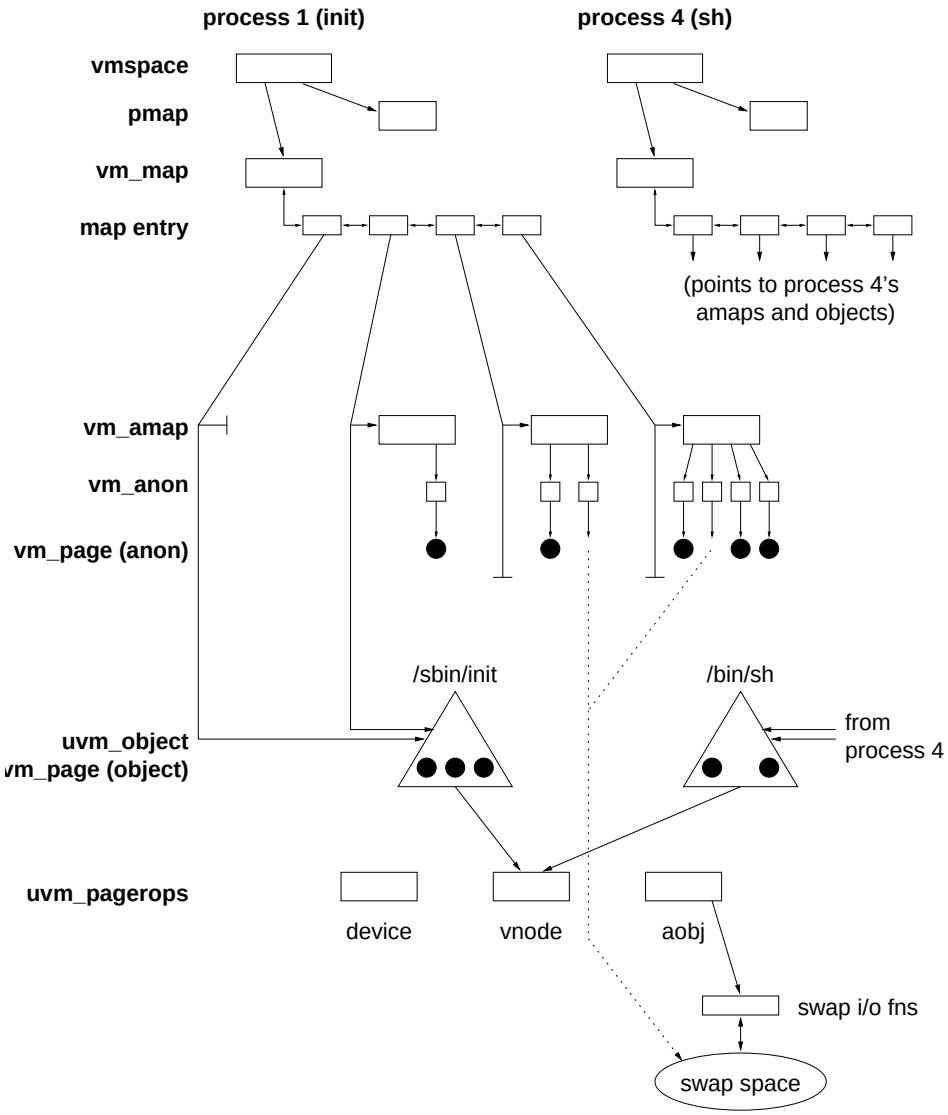
UVM został zaprojektowany i zaimplementowany przez Charles'a Cranor'a. Cranor prezentuje UVM w swojej pracy doktorskiej (patrz literatura) w 1998roku.

W dalszej części rozdziału opisane będą różnice między VM a UVM, w szczególności będzie zaprezentowana realizacja fork nie obciążona problemem „swap memory leak"

3.2 Co zostało z VM

UVM stosuje bez żadnych zmian (poza drobnymi implementacyjnymi) warstwę zależną od sprzętu z BSD 4.4VM. Wszelkie zmiany dotyczą warstwy MI. W warstwie MI bez zmian jednak zostały struktury *vm_space*, *vm_map* oraz *vm_mapentry* (poza tym, że wskazuje na inne obiekty, patrz niżej).

3.3 Co się zmieniło



Najbardziej w oczy rzuca się wprowadzenie nowych rodzajów obiektów: vm_amap i vm_anon. Oba służą do obsługi tzw. pamięci anonimowej. Pamięć anonimowa to taka, która nie jest związana z konkretnym plikiem, a tym samym wymieniana jest w „swap-ie”. Przykłady pamięci anonimowej to:

– stos/sterta procesu

– obszary pamięci dzielonej

– zmienione strony przy „copy–on–write”

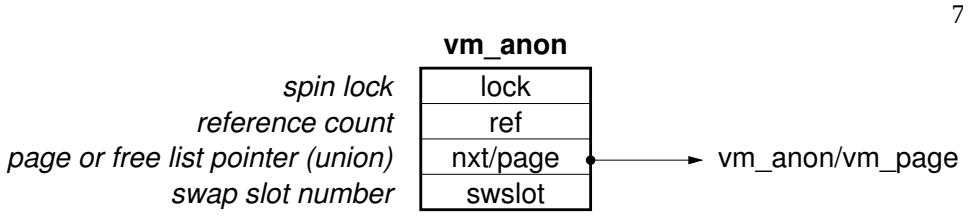
W BSD 4.4 obsługą pamięci anonimowej „zajmowały się” struktury vm_object w połączeniu z mechanizmem obiektów przesłaniających. W UVM zrezygnowano z tego mechanizmu, zastępując go strukturami *vm_amap* i *vm_anon*.

Poszukiwanie strony, do której nastąpiło odwołanie odbywa się teraz w następujący sposób:

1. Odnajdywana jest odpowiednia pozycja w *map_entry*
2. map_entry może wskazywać albo na vm_object albo na vm_amap albo na obie struktury
3. Najpierw strony szuka się w strukturach vm_amap, następnie w vm_object

Jak widać vm_amap w połączeniu z vm_anon gra podobną rolę jak obiekty przesłaniania w BSD VM. Jednak w UVM nie powstają żadne łańcuchy obiektów - mamy po prostu dwie warstwy.

3.3.1 vm_anon



Struktura ta jest otoczką dla pojedynczej strony pamięci anonimowej. Zawiera lock, licznik odwołań, wskaźnik do opisywanej strony, oraz slot w przestrzeni wymiany jeśli opisywana strona została wyrzucona z pamięci fizycznej (wpp. ten slot ma numer 0). Kiedy strona jest zwolniona obiekt *vm_anon* nie jest niszczony, ale dołącza do listy wolnych obiektów typu *vm_anon*. Lista ta jest zbudowana ze wskaźników do opisywanej strony (bo dla wolnych vm_anon jest on niepotrzebny) - zrealizowane jest to w ten sposób, że pole opisujące ten wskaźnik jest unią w C.

3.3.2 vm_amap

Jak wspomniano wyżej *vm_mapentry* wskazuje na vm_amap. Struktura ta opisuje pamięć anonimową zawartą w obszarze adresowym opisywanym przez *vm_mapentry*. Z abstrakcyjnego punktu widzenia *vm_amap* to tablica, która na poszczególnych pozycjach zawiera albo *vm_anon* albo pustą pozycję (pozycje odpowiadają stronom). W rzeczywistości *vm_amap* zawiera dodatkową listę z wypełnionymi pozycjami (oraz struktury do synchronizacji tablicy i listy) - lista ta pozwala na szybkie wykonywanie operacji na wszystkich obiektach *vm_anon*

3.3.3 uvm_object

Struktura uvm_object została okrojona w UVM - nie ma już potrzeby pamiętania obiektów przesłaniania. vm_object zawiera tylko listę stron, które do niego należą, licznik odwołań oraz wskaźnik do obiektu, gdzie ma być wymieniany.

3.4 Realizacja fork

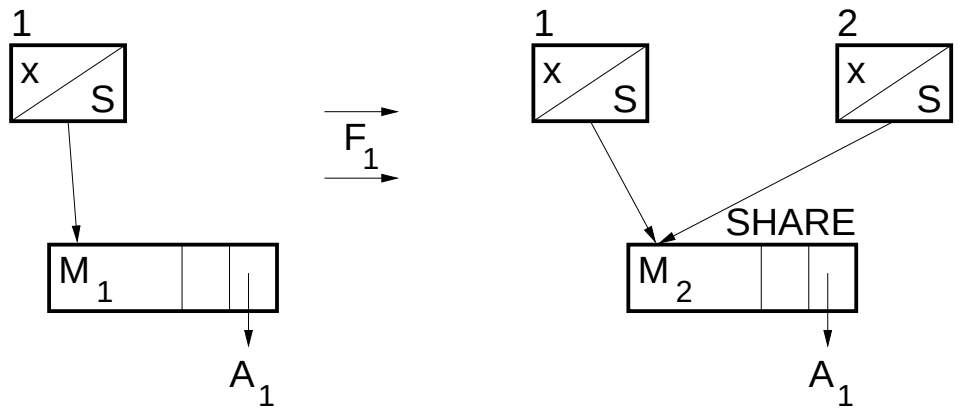
Wykonanie fork polega na przejrzeniu wszystkich *map_entry* procesu macierzystego oraz utworzeniu w procesie potomnym odpowiadającego mu *map_entry*. Mogą tu być następujące przypadki

1. Przestrzeń adresowa ma być niedostępna dla potomka - w tym wypadku nie trzeba nic robić
2. Przestrzeń adresowa ma być współdzielona
3. Przestrzeń adresowa ma być kopiowana (na zasadzie „copy–on–write”)

To czy dany fragment pamięci będzie współdzielony czy „copy–on–write” jest ustawione w odpowiednim polu struktury *map_entry*.

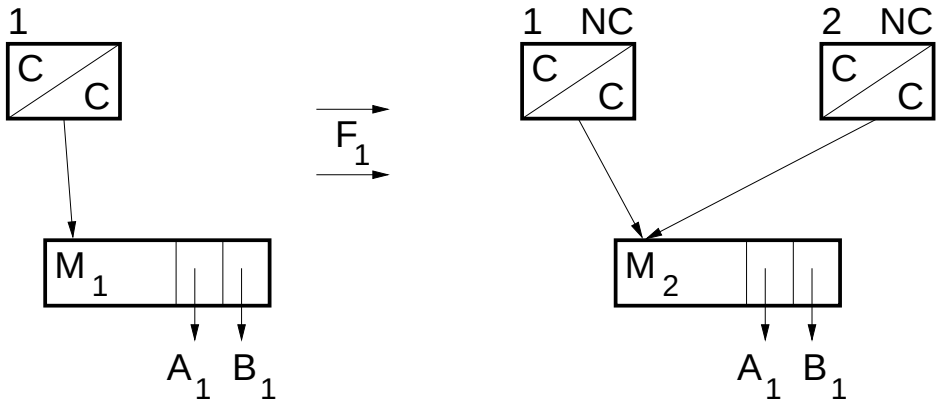
3.4.1 Przestrzeń adresowa współdzielona

Ten przypadek pokazany jest na rysunku poniżej. Tutaj nowy proces zwyczajnie „podpina się” pod strukturę *vm_amap*. Zauważmy, że licznik odwołań do *vm_amap* zwiększył się o jeden, a tym samym otrzymała ona atrybut SHARE. Anon A nie został zmieniony, ale widzą go oba procesy (dzięki temu, że oba mapują tę samą strukturę *vm_amap*. Oprócz tego proces potomny mapuje także obiekt *uvm_object* (jeśli taki jest w *map_entry* procesu macierzystego.



3.4.2 Przestrzeń adresowa ma być kopiowana

Pierwszy krok w tym wypadku jest podobny jak poprzednio - proces potomny mapuje się na *vm_map* procesu macierzystego. Tym razem zamiast ustawiać mapę jako dzieloną - ustawiane są flagi „needs-copy” w obu procesach, a strony należące do procesu macierzystego są ustawiane jako read-only, żeby próba zapisu spowodowała page-fault (w procesie potomnym page-fault wystąpi zawsze bo nie ma zmapowanych żadnych stron)



Załóżmy teraz, że proces macierzysty próbuje zapisać do anon A, spowoduje to pagefault. W wyniku jego obsługi wykonane zostaną następujące kroki:

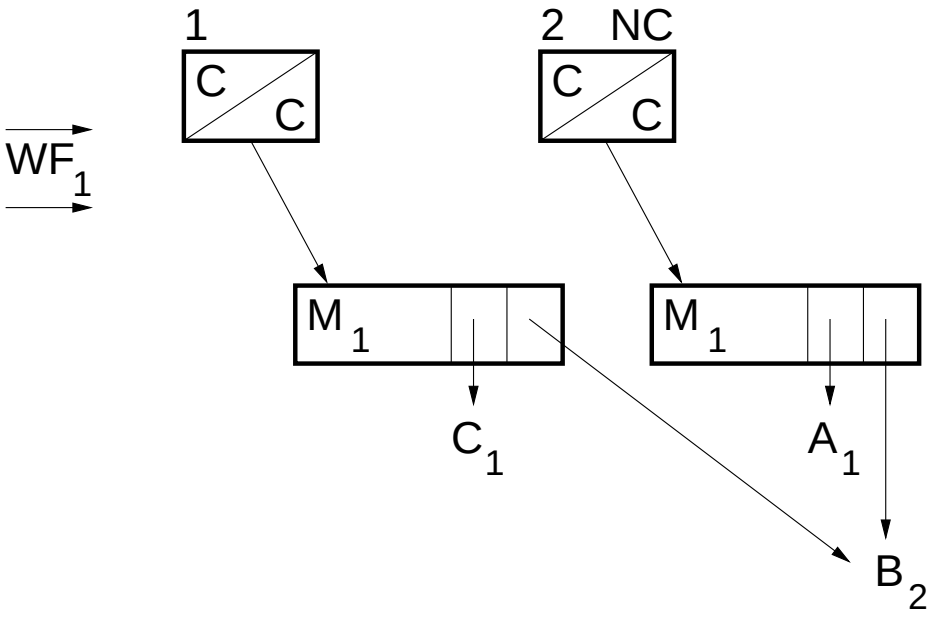
- Stworzony zostanie nowy *vm_anon* - na rysunku C
- Zostanie skopiowane *vm_amap*
- Skasowana zostanie flaga „needscopy” w procesie macierzystym

Zauważmy, że proces potomny ma nadal ustawionę flagę „needscopy”, ale to nie jest żaden problem, ponieważ kopiowanie *vm_amap* wykonywane jest tylko wtedy, gdy licznik odwołań do amap przekracza 1 (wpp kopiowanie oczywiście nie ma sensu).

3.4.3 Szczególne przypadki

Powyższy opis wystarcza dla większości wypadków stosowania fork. Jednak są szczególne przypadki kiedy wymagana jest nieco inna procedura (sprowadza się ona zwykle do wcześniejszego wykonania kopiowania *amap* i wyzerowania flagi „needs–copy”. Te przypadki to:

- Proces macierzysty ma ustawioną flagę „needs–copy” przy forkowaniu „współdzielonym”



– Forkowanie procesu, który dzieli pamięć z innym (jego *vm_amap* ma licznik odwołań większy od 1)

Rozwiązanie tych problemów jest łatwe i nie wnosi niczego nowego do zasady działania UVM. Po szczegóły odsyłam do literatury.

3.5 Podsumowanie

Powyżej przedstawiłem najbardziej spektakularne usprawnienie UVM w stosunku do VM. Inne ulepszenia oraz dodatkowe możliwość UVM znacznie podnoszą wydajność zarządcy pamięci wirtualnej i samego systemu. Przekonać się o tym mogą użytkownicy NetBSD (gdzie zaimplementowano UVM). Otóż okazuje się, że wprowadzenie UVM powoduje znaczący wzrost wydajności systemu (nawet o kilkanaście procent).

4 Bibliografia

1. `http://www.netbsd.org` - strona domowa systemu NetBSD, w którym zostało zaimplementowane UVM. Można z niej między innymi ściągnąć źródła kernel-a z UVM oraz poczytać dokumentację
2. `http://www.netbsd.org/Documentation/kernel/uvm.html` - dokumentacja UVM w NetBSD. Zawiera między innymi link do pracy doktorskiej Chuck'a Cranor'a wprowadzającej UVM, z której korzystałem przy przygotowywaniu tej prezentacji.
3. `http://www.ccrc.wustl.edu/pub/chuck/tech/uvm` - strona domowa UVM, zawiera FAQ oraz linki do dokumentacji

4. `http://www.usenix.org/events/usenix99/full_papers/cranor/cranor.pdf` - artykuł pokrótce opisujący UVM. Można powiedzieć, że jest to wspomniana wyżej praca doktorska - w pigułce.