# Krótki przegląd systemu zarządzania pamięcią wirtualną w BSD

Michał Kozłowski

10 grudnia 2002

## Wprowadzenie

W tej prezentacji omówione zostaną podsystemy zarządzania pamięcią w systemach BSD:

- Tradycyjny moduł z wersji BSD 4.4
- UVM

## 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)

## Warstwa zależna od sprzętu

- Udostępnia strukturę *pmap* będącą abstrakcją jednopozio
  tablicy stron

- Do zarządzania strukturą *pmap* służą następu-
  jące funkcje:

  **pmap_enter** tworzy nowe wiązania między wirtu-
  alnym indeksem, a fizyczną stroną ze wskazanymi
  atrybutami ochrony.

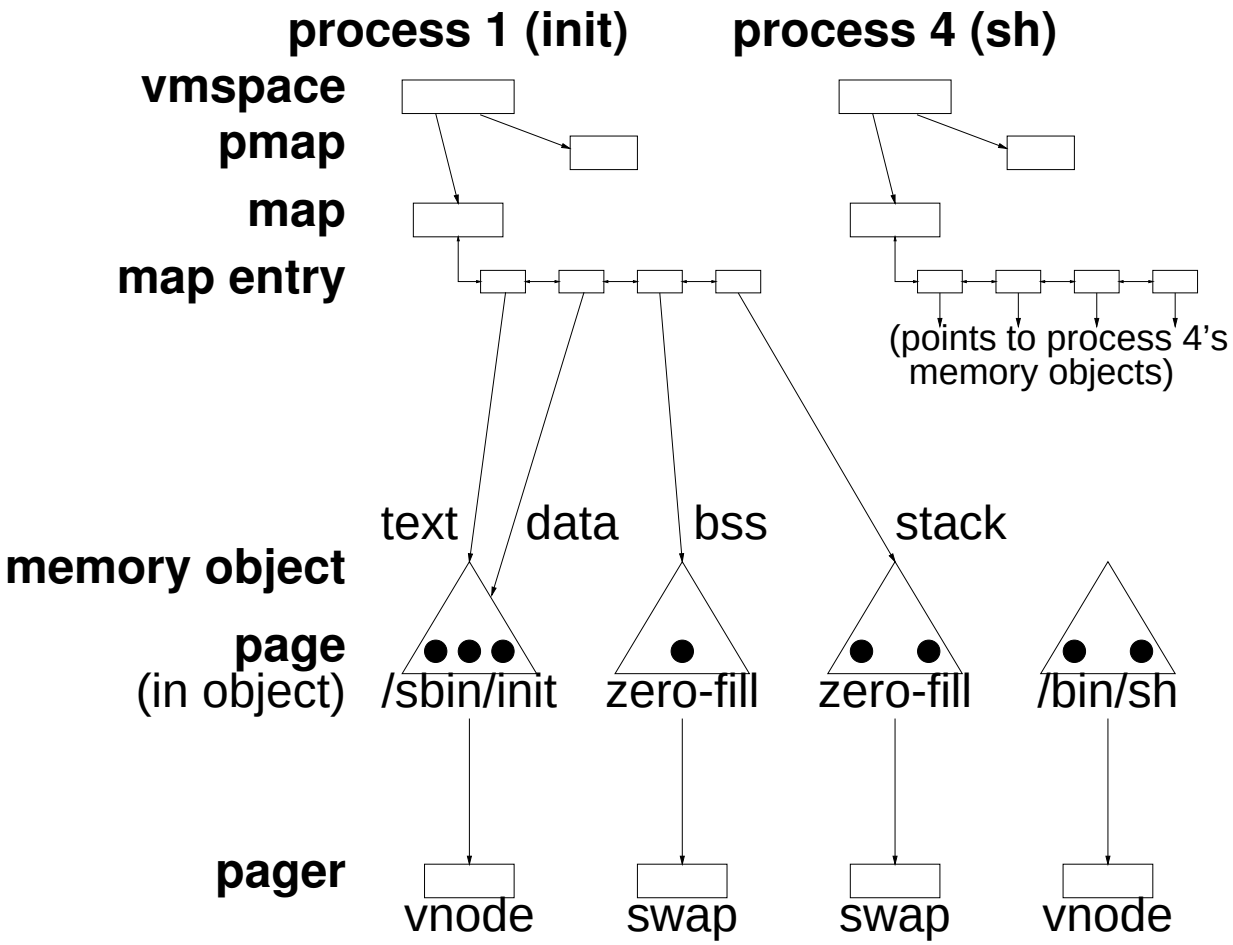
  **pmap_remove** - usuwanie wiązania. Wywoływana
  przy zwalnianiu stron

  **pmap_protect** - ustawianie atrybutów ochrony strony

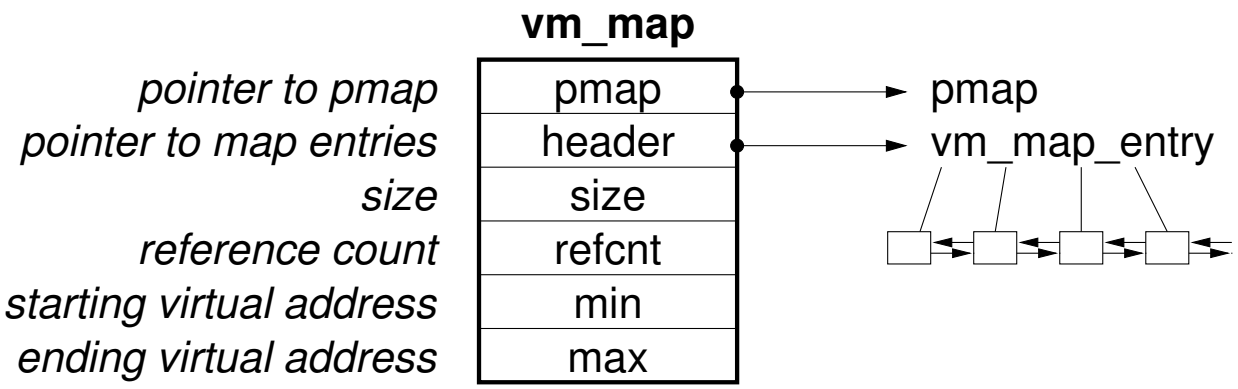
  **pmap_reference,pmap_modified** - odczytywanie
  bitów dostępu/modyfikacji

  **pmap_clear_reference,pmap_celar_modify** - ze-
  rowanie bitów dostępu/modyfikacji

# Warstwa MI


process 1 (init)
process 4 (sh)
vmspace
pmap
map
map entry
(points to process 4's memory objects)
text
data
bss
stack
memory object
page
(in object)
/sbin/init
zero-fill
zero-fill
/bin/sh
pager
vnode
swap
swap
vnode

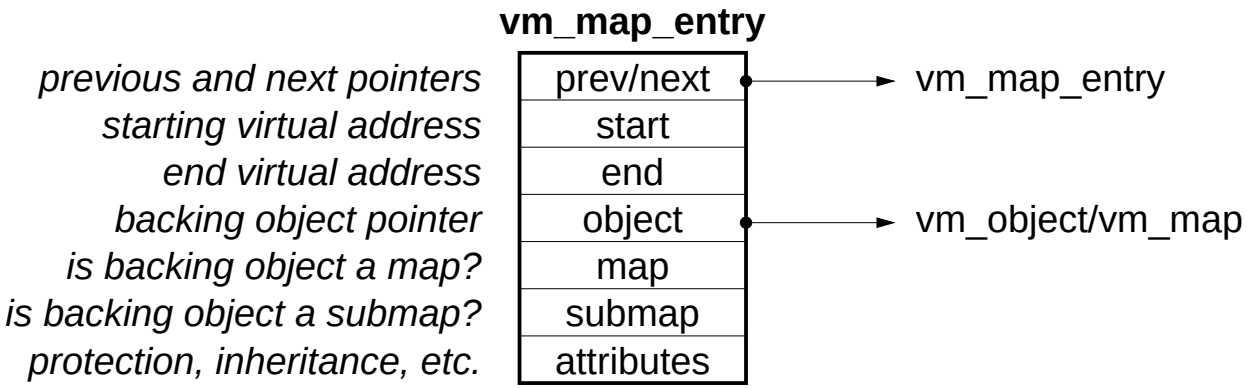
## vm_map




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

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

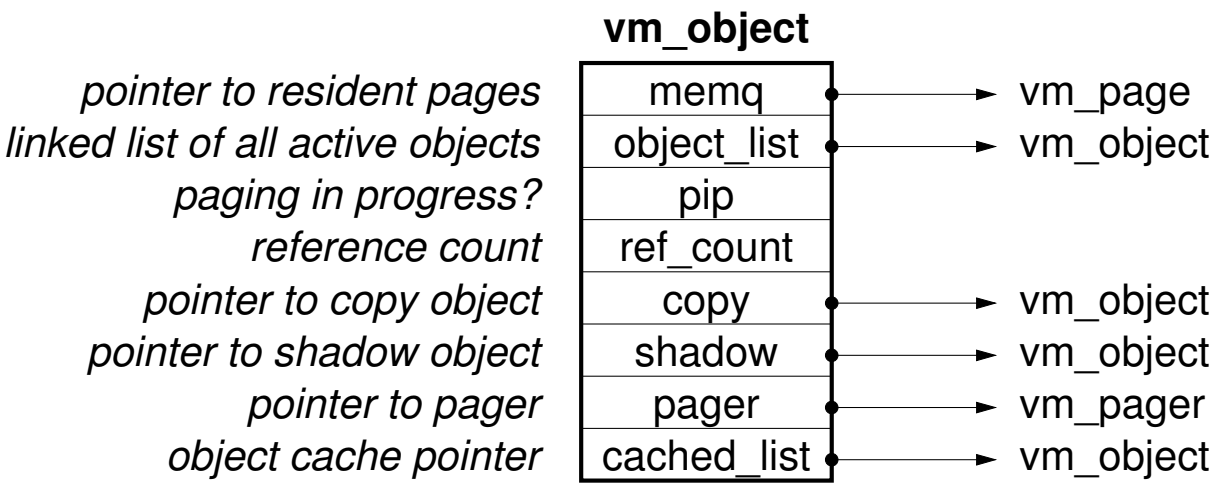
## vm_map_entry




- opisuje przyporządkowanie konkretnego fragmentu wirtualnej przestrzeni adresowej do pewnego obiektu.

- *map_entry* są zorganizowane w dwukierunkową listę posortowaną względem adresu początku opisywanego obszaru

# vm_object

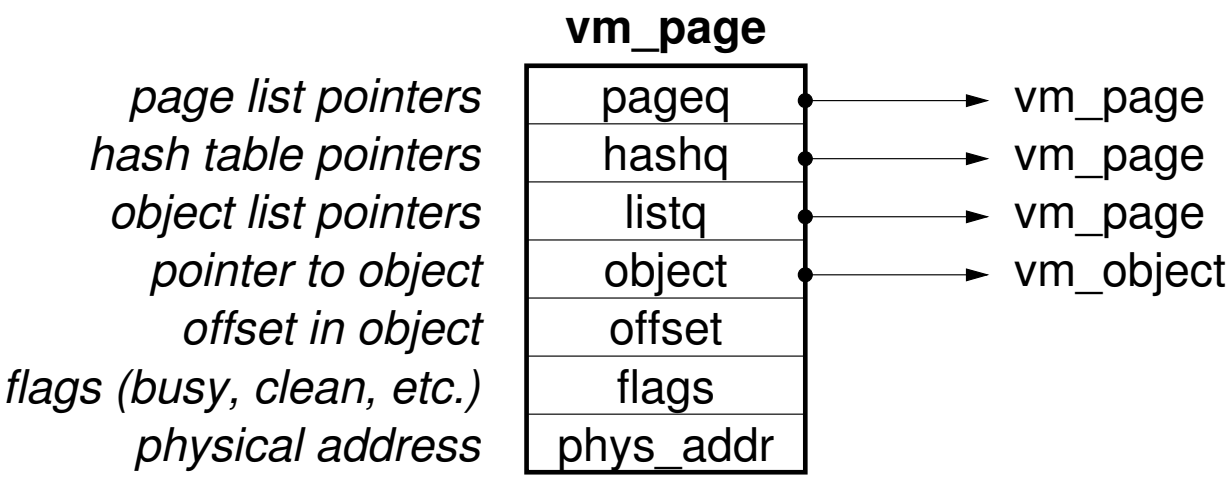



- Na obiekt składa się pewna (dowolna) liczba stron (opisywanych przez strukturę *vm_page*

- Strony obiektu są wymieniane przy pomocy *vm_pager*

- Obiekty są cache'owane, żeby można je było użyć ponownie bez konieczności ponownego ładowania z dysku (przykład obiekt reprezentujący `/bin/ls`

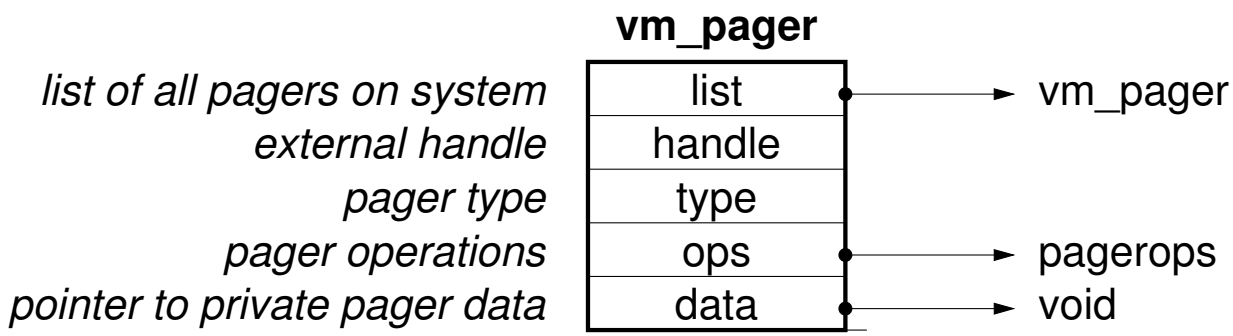
## vm_page




- *vm_page* reprezentuje pojedynczą stronę pamięci fizycznej.

- Struktura ta zawiera odwołanie do obiektu, do którego należy strona, jej fizyczny adres oraz atrybuty (ochrona, bity modyfikacji itd.)

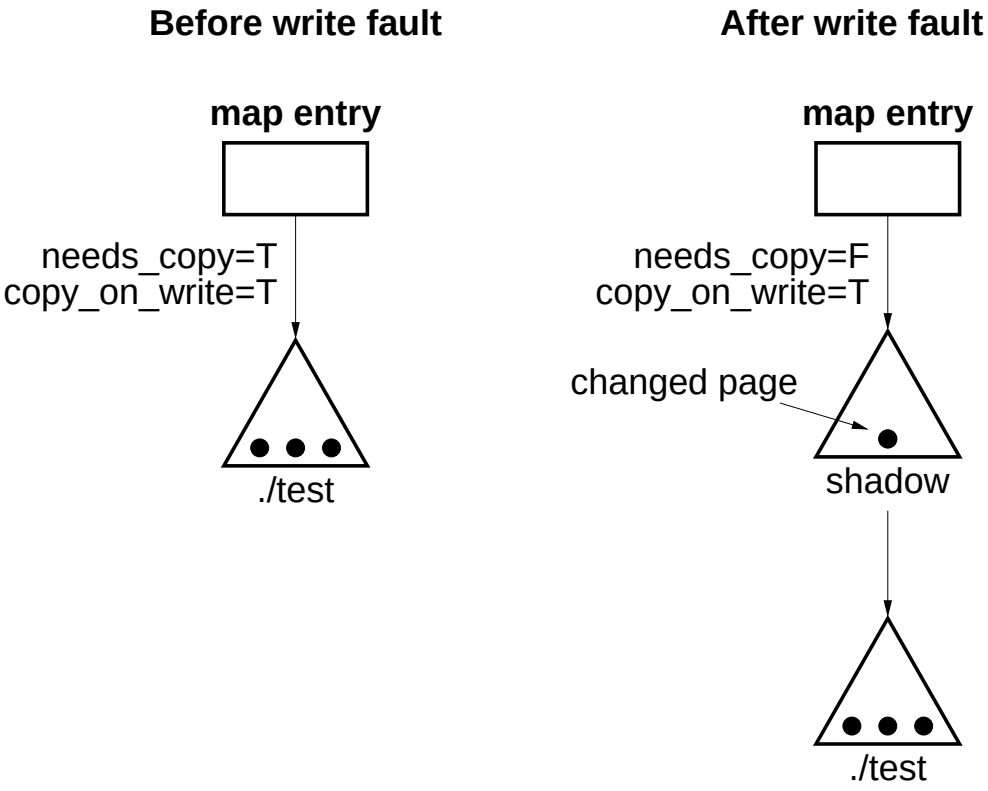
## vm_pager




- Reprezentuje obiekt, na którym dokonuje się wymiany stron.

- Może to być np. *vnode* reprezentujący plik albo *swap* reprezentujący przestrzeń wymiany.

# Copy–on–write i obiekty przesłaniające

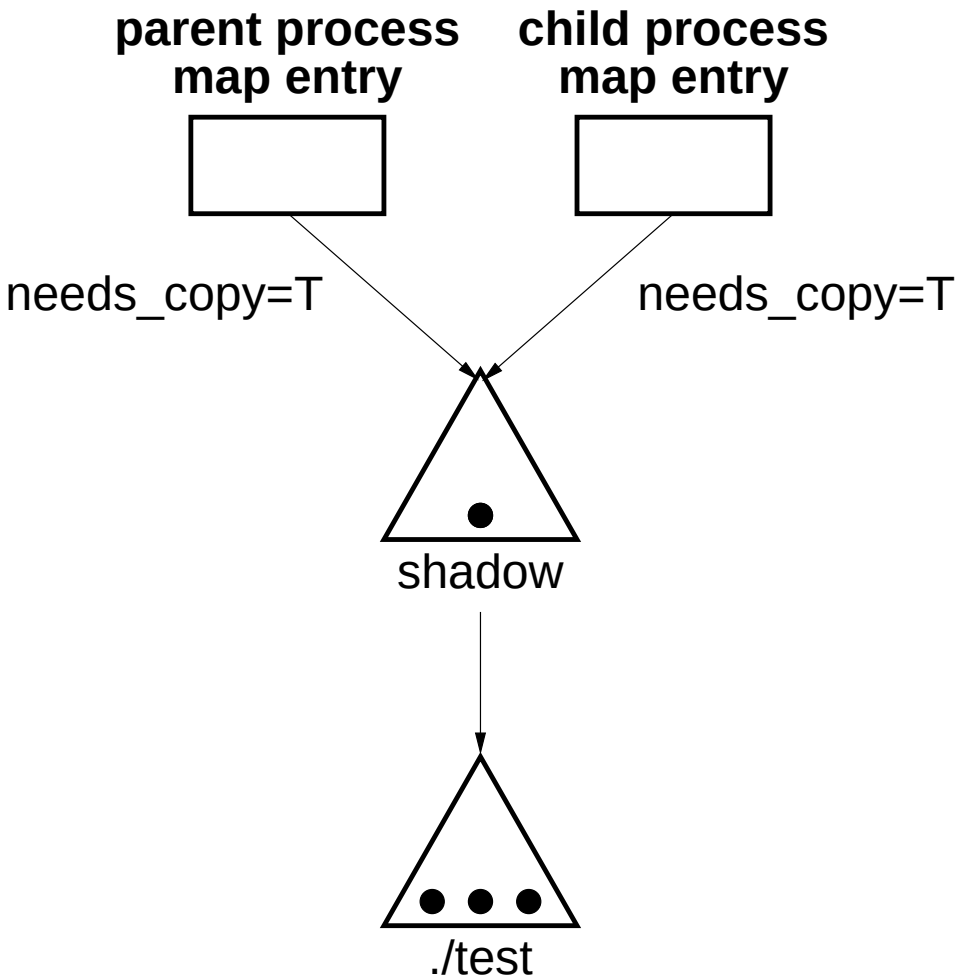



- Flagi needs–copy i copy–on–write
- Obiekt przesłaniający, zawiera strony które zostały zmodyfikowane
- Obiekty tworzą łańcuch

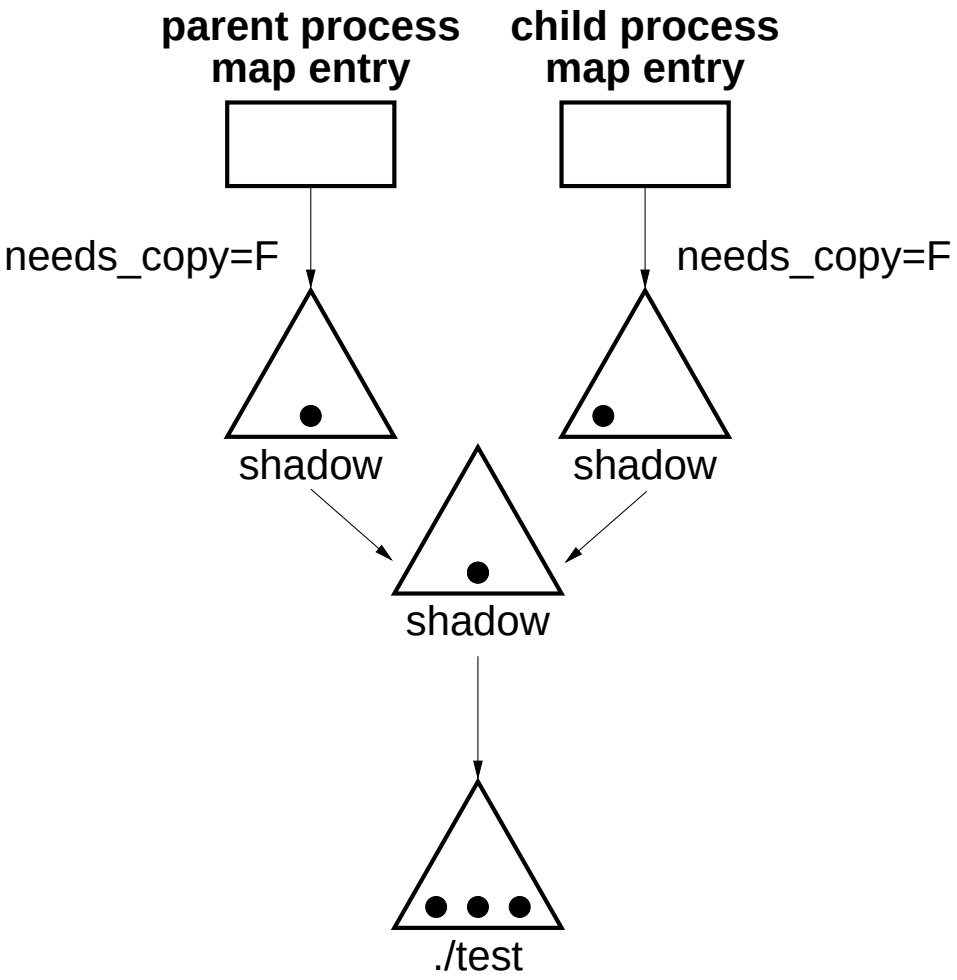
## Realizacja fork

- Mechanizm *copy–on–write* jest intensywnie wykorzystywany, przy realizacji funkcji systemowej *fork*.

- Samo wywołanie fork nie powoduje kopiowania



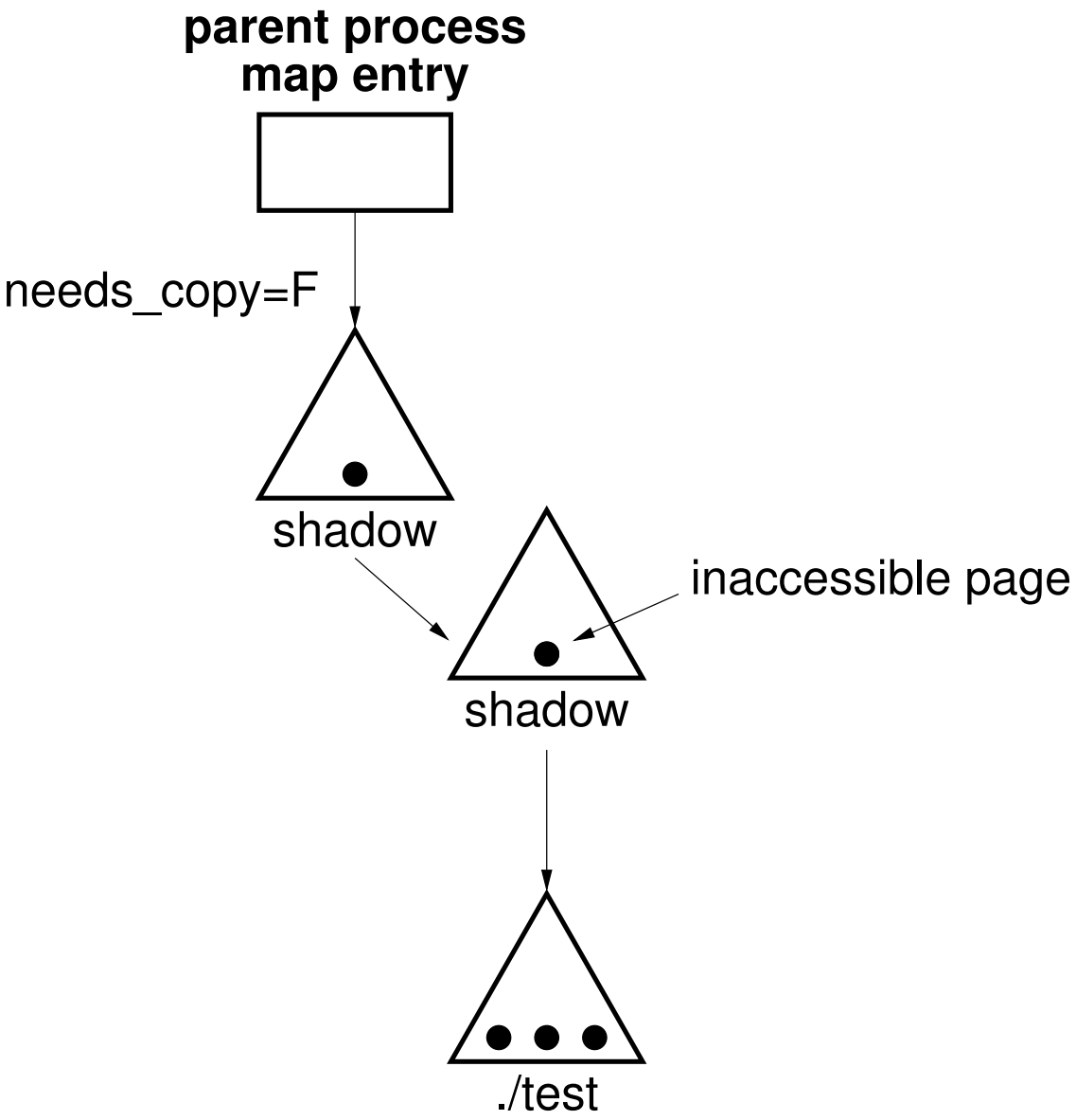
- Próba zapisu powoduje utworzenie obiektu przesłaniającego



## Problem długich łańcuchów i „wycieków” pamięci

- Powtarzanie fork’a powoduje wydłużanie łańcucha obiektów

- Zakończenie procesu potomnego może powodować „swap memory leak”


parent process
map entry
needs_copy=F
shadow
inaccessible page
shadow
./test

## UVM

- UVM to nowy system pamięci wirtualnej, zastępuje VM stosowany w 4.4 BSD. UVM przez uproszczenie struktur danych (np. wyeliminowanie

- ulepszony mechanizm copy–on–write przez całkowite jego przeprojektowanie

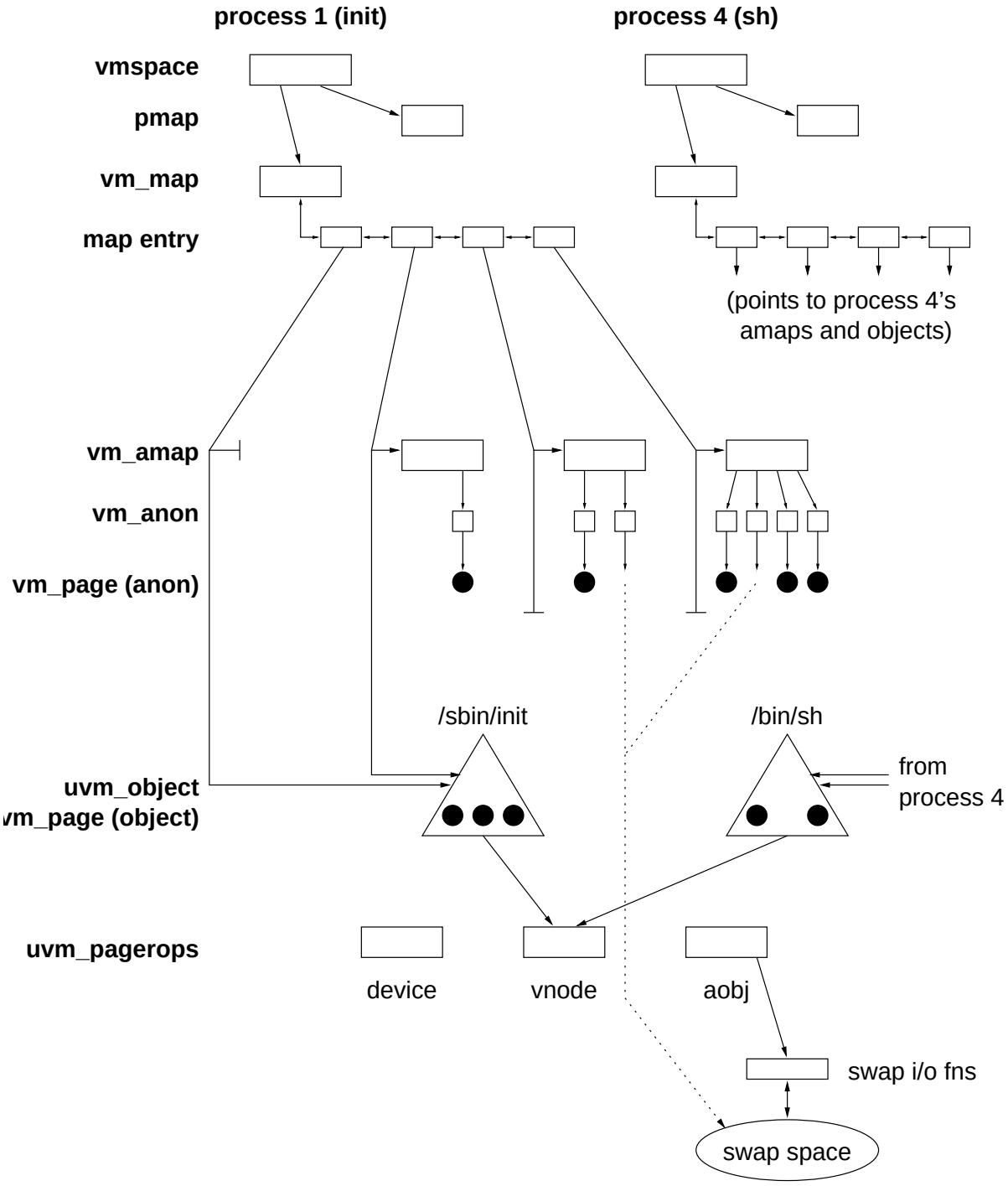
- zaprojektowany i zaimplementowany przez Charles'a Cranor'a (publikacja pracy doktorskiej w 1998 roku)

## Co zostało z VM

- Warstwa zależna od sprzętu

- W warstwie MI praktycznie bez zmian zostały struktury *vm_space*, *vm_map* oraz *vm_mapentry*

# Co się zmieniło



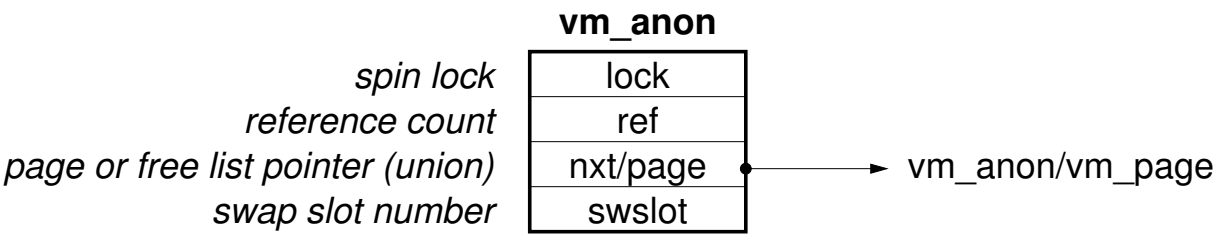
- Wprowadzono vm_amap i vm_anon - służą do obsługi pamięci anonimowej

- Przykłady pamięci anonimowej
  - stos/sterta procesu
  - obszary pamięci dzielonej
  - zmienione strony przy „copy–on–write”

- Algorytm poszukiwania strony
  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

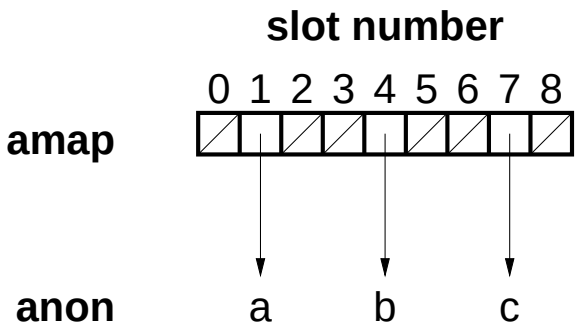
# vm_anon

7




- Otoczka dla pojedynczej strony pamięci anonimowej
- Podwójna rola wskaźnika do stron

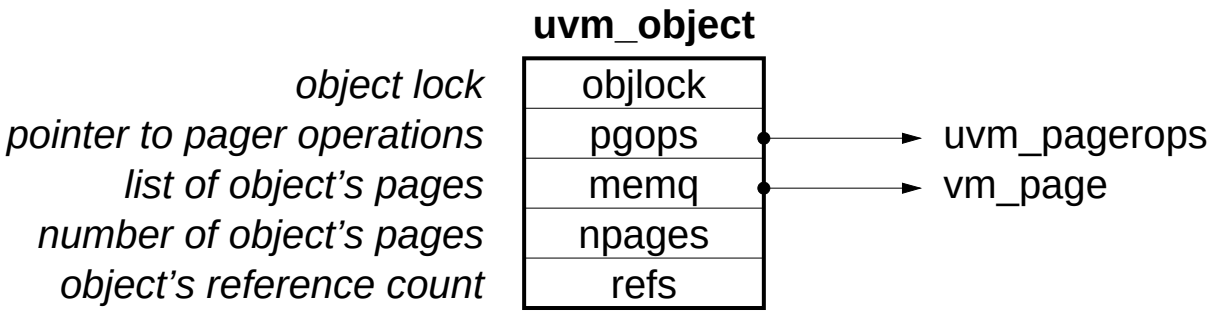
## vm_amap




- *vm_amap* to tablica, która na poszczególnych pozycjach zawiera albo *vm_anon* albo NULL (pozycje odpowiadają stronom).

- Lista zajętych pozycji

# uvm_object




- Okrojony w stosunku do VM
- Nie ma obiektów przesłaniających

## Realizacja fork w UVM

- Wykonanie fork polega na przejrzeniu wszystkich *map_entry* procesu macierzystego oraz utworzeniu w procesie potomnym odpowiadającego mu *map_entry*.

- Mogą tu być następujące przypadki

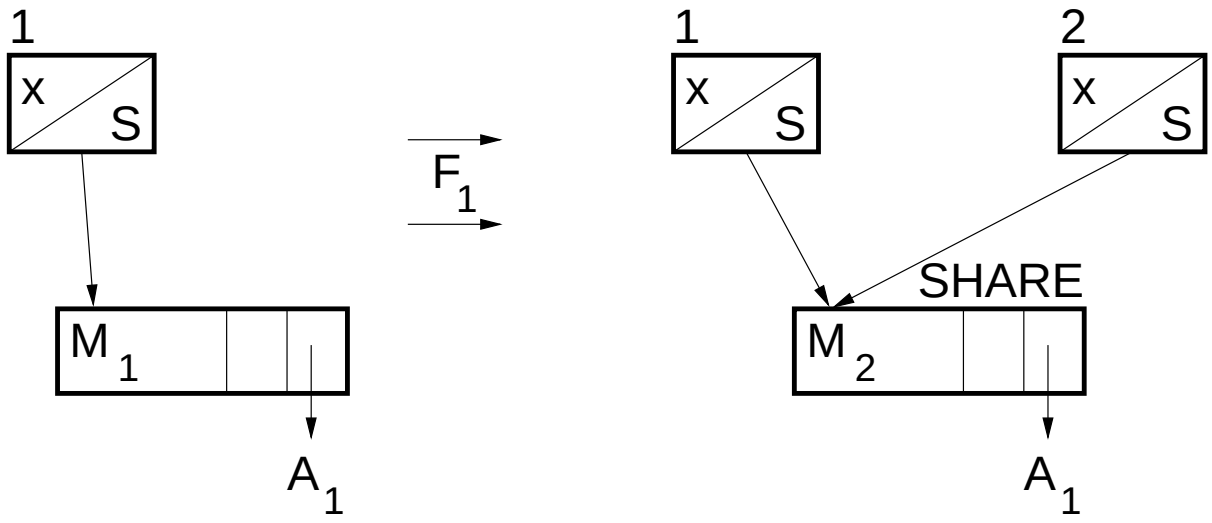
  1. Fragment przestrzeni adresowej ma być niedostępny dla potomka - w tym wypadku nie trzeba nic robić

  2. Fragment przestrzeni adresowej ma być współdzielony

  3. Fragment przestrzeni adresowej ma być kopiowany (na zasadzie „copy–on–write”)

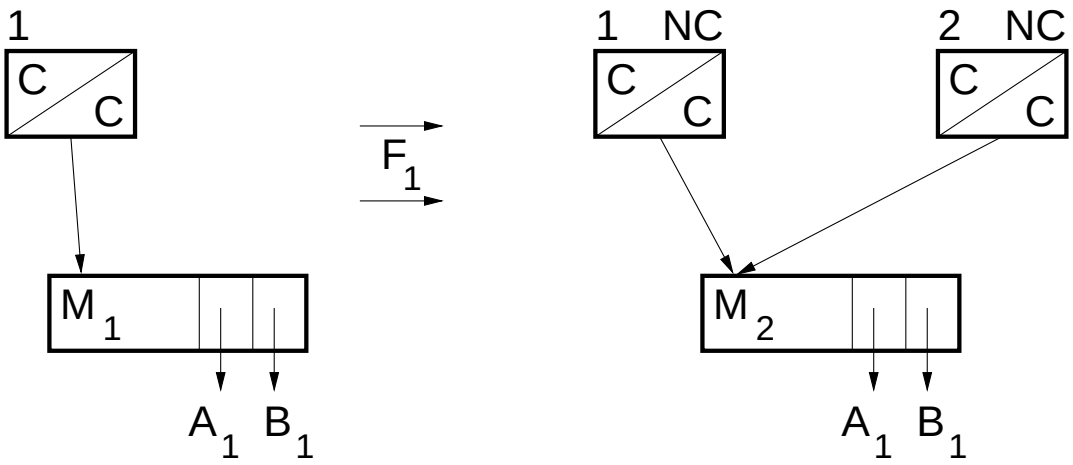
- To czy dany fragment pamięci będzie współdzielony czy „copy–on–write” jest ustawione w odpowiednim polu struktury *map_entry*.

## Przestrzeń adresowa współdzielona



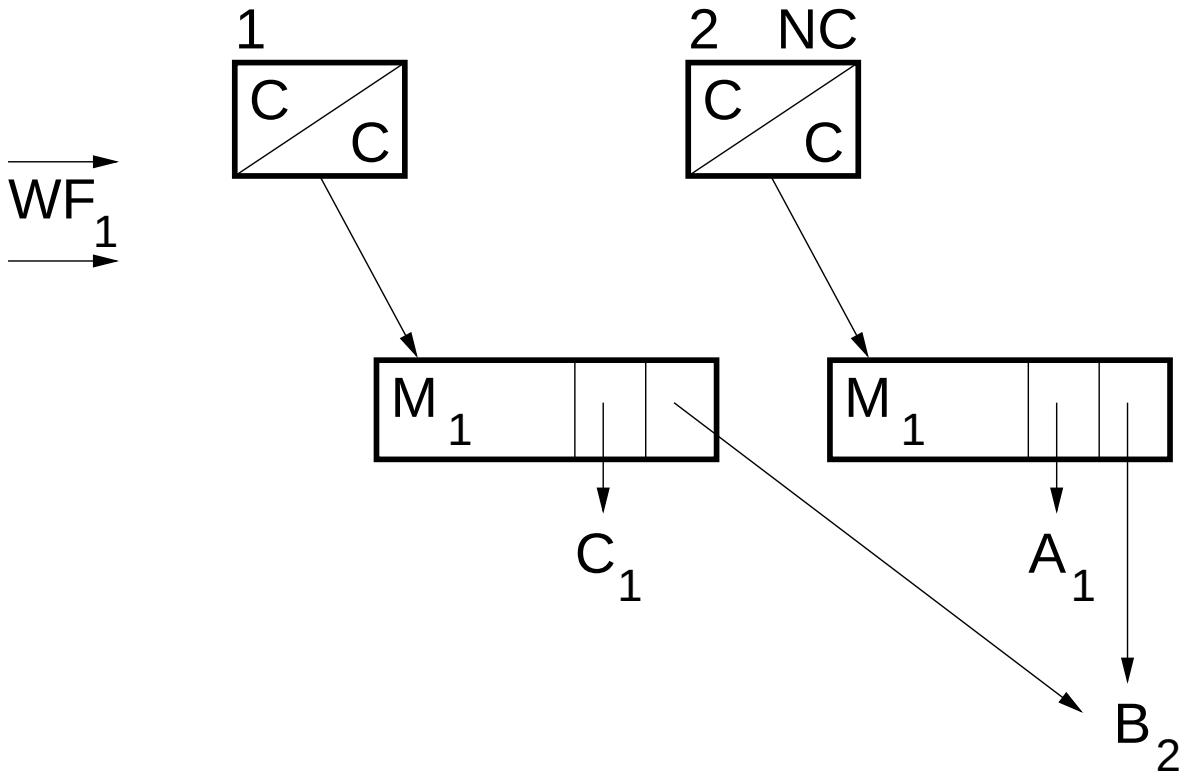

- Nowy proces„podpina się” pod strukturę *vm_amap*
- Licznik odwołań do *vm_amap* zwiększył się o jeden, otrzymała ona atrybut SHARE.
- Anon A nie został zmieniony
- Widzą go oba procesy (dzięki temu, że oba mapują tę samą strukturę *vm_amap*.

## Przestrzeń adresowa ma być kopiowana cz.1




- Pierwszy krok jak poprzednio
- Zamiast ustawiać mapę jako dzieloną - ustawiane są flagi „needs-copy” w obu procesach,
- Strony należące do procesu macierzystego są ustaw-iane jako read-only, żeby próba zapisu spowodowała page-fault
- W procesie potomnym page-fault wystąpi zawsze bo nie ma zmapowanych żadnych stron

## Przestrzeń adresowa ma być kopiowana cz.2




- 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

- Proces potomny ma nadal ustawioną flagę „needscopy”

- Nie jest to problem, ponieważ kopiowanie *vm_amap* wykonywane jest tylko wtedy, gdy licznik odwołań do amap przekracza 1

## Podsumowanie

- Inna realizacja copy–on–write to największa różnica w stosunku do BSD VM
- UVM zawiera pewne operacje niedostępne w BSD VM, które również wpływają na poprawę wydajności
- Korzyści z wprowadzenia UVM obserwować mogą użytkownicy NetBSD
- Wzrost wydajności widoczny zwłaszcza w sytuacjach, kiedy brakuje pamięci fizycznej

## Bibliografia


1. *http://www.netbsd.org* - strona domowa systemu NetBSD. 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.

5. *http://rainbow.mimuw.edu.pl/mk189412/uvm.pdf*