Algorytm **brk**

1 Wstęp

Algorytm **brk** służy do zmiany rozmiaru pamieci przeznaczonej na obszar danych procesu.

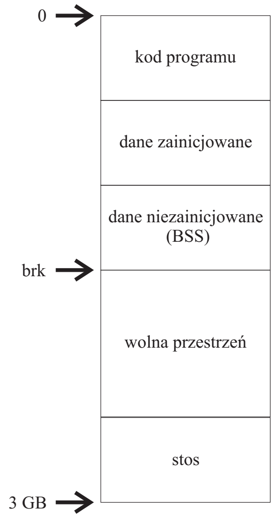
Autor: Paweł Daniluk

2 Mapa pamięci wirtualnej procesu i opisujące ją struktury danych

Od początku istnienia komputerów zapotrzebowanie na pamięć przekraczało rozmiar dostępnych zasobów. W wielozadaniowych systemach operacyjnych doskonałym mechanizmem "powiększenia"dostępnych zasobów jest **pamięć virtualna** (ang. *virtual memory*). W systemie Linux każdy proces dysponuje 3 GB przestrzeni adresowej, w której odwzorowywane są wykorzystywane przez proces obszary pamięci. Mapa pamięci procesu składa się z:

- obszaru kodu programu,
- obszaru danych,
- obszaru BSS[1] (niezainicjowanych danych),
- wolnej przestrzeni adresowej wykorzystywanej do mapowania plików i pamięci dzielonej,
- stosu.

[1] Nazwa BSS jest reliktem historycznym i pochodzi ze starego operatora asemblerowego "Block Started by Symbol



Rysunek 1: Mapa pamięci wirtualnej procesu

Obszar kodu programu i danych wczytywane są z pliku wykonywalnego. Obszar BSS jest przeznaszony na `stertę` (ang. *heap*) procesu. Do opisu pamięci wirtualnej procesu służą następujące struktury danych:

- `mm_struct` - główna struktura opisująca pamięć procesu, zawiera wskaźniki do listy i drzewa AVL struktur `vm_area_struct`, wskaźniki do podstawowych obszarów etc.
- `vm_area_struct` - struktura opisująca pojedynczy zmapowany obszar

Struktury `vm_area_struct` są powiązane w drzewo AVL.

3 Cel algorytmu brk

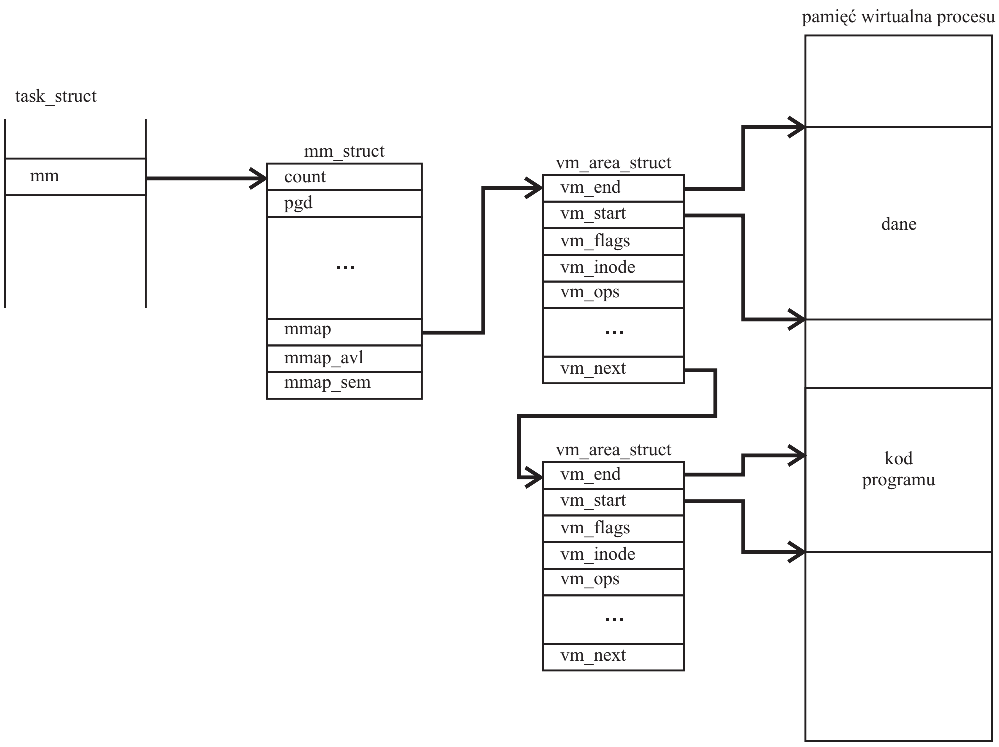
Jak widać na rys. 1 tytułowe **brk** to wskaźnik do pierwszej komórki pamięci za blokiem BSS. Algorytm **brk** służy do zmieniania tego wskaźnika, czyli do powiększania lub zmniejszania obszaru BSS. Algorytm jest wykonywany przy wywołaniu funkcji `brk` i `sbrk` oraz może być wywoływany przy okazji funkcji `malloc` i `free`.

4 Wywołanie systemowe i funkcja `sys_brk`

4.1 Ogólny opis

Wywołanie systemowe `sys_brk` powoduje wykonanie funkcji `sys_brk`.

```
unsigned long sys_brk(unsigned long brk)
```



Rysunek 2: Schemat struktur danych opisujących pamięć wirtualną procesu

Parametrem funkcji `sys_brk` jest nowa wartość wskaźnika `brk`. Funkcja zwraca wartość jaką `brk` przybrało po jej wykonaniu[2].

4.2 Schemat działania funkcji `sys_brk`

Jak widać funkcja `sys_brk` wykonuje jedynie sprawdzenie poprawności parametrów oraz dostępnej pamięci i jeżeli sprawdzenia wypadną pozytywnie zleca wykonanie samego przydziału innej funkcji.

4.3 Funkcje `find_vma`, `find_vma_intersection`

Funkcja `find_vma` zwraca strukturę `vm_area_struct` opisującą pierwszy obszar pamięci (w danej mapie pamięci) zaczynający się za podanym adresem. Jeżeli taki obszar nie istnieje zwracana jest wartość `NULL`.

```
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
```

Funkcja przeszukuje drzewo AVL obszarów pamięci (`mm->mmap_avl`) lub, jeżeli ono nie istnieje, listę obszarów pamięci (`mm->mmap`).

[2]Czyli wartość parametru jeżeli operacja się powiodła i starą wartość `brk` w przeciwnym przypadku

Algorytm 1 sys_brk

Dane wejściowe: `brk`

Wynik: 0 lub kod błędu

```
opuszczenie semafora mm->map_sem i blokada jądra
jeśli brk < koniec obszaru kodu to
  porażka
koniec jeśli
jeśli brk <= mm->brk to
  zwolnienie pamięci (funcja do_munmap)
  koniec
koniec jeśli
jeśli przekroczenie limitu podanego w rlim[RLIMIT_DATA] to
  porażka
koniec jeśli
jeśli obszar, ktory chcemy przydzielić jest częściowo użyty do innych mapowań (funkcja
find_vma_intersection) to
  porażka
koniec jeśli
jeśli brak wystarczającej ilości wolnej pamięci (funkcja vm_enough_memory)  to
  porażka
koniec jeśli
zamapowanie pamięci (funkcja do_mmap)
podniesienie semafora i odblokowanie jądra
```

Funkcja `find_vma_intersection` wyszukuje w danej mapie pamięci obszar przecinający się z zadanym zakresem.

```
struct vm_area_struct * find_vma_intersection(struct mm_struct * mm,
     unsigned long start_addr, unsigned long end_addr)
```

Aby znaleźć obszary położone w okolicy podanego zakresu, funkcja wywołuje funkcję `find_vma`.

4.4 Funkcja `vm_enough_memory`

Funkcja `vm_enough_memory` sprawdza czy rozmiar wolnej pamięci (liczba wolnych stron) jest większy od podanej wielkości. Jeżeli tak funkcja zwraca 1, w przeciwnym przypadku zwraca 0.

```
int vm_enough_memory(long pages)
```

Liczba wolnych stron obliczana jest według wzoru:

```
buffermem >> PAGE_SHIFT + page_cache_size + nr_free_pages + nr_swap_pages
- (page_cache.min_percent + buffer_mem.min_percent + 2) * num_physpages / 100
```

Gdzie:

- `buffermem ≫ PAGE_SHIFT` - liczba stron przeznaczonych na **podręczną pamięć buforową** (ang. *buffer cache*)
- `page_cache_size` - liczba stron przeznaczonych na **podręczną pamięć stron** (ang. *page cache*)
- `nr_free_pages` - liczba wolnych ramek pamięci
- `nr_swap_pages` - liczba wolnych stron w urządzeniach wymiany stron
- `num_physpages` - liczba wszystkich ramek pamięci
- `buffer_mem.min_percent` i `buffer_mem.min_percent` - minimalny procentowy rozmiar podręcznej pamięci buforowej i podręcznej pamięci stron

Jeżeli zmienna `sysctl_overcommit_memory` jest różna od zera, to dopuszcza się przydzielenie większej ilości pamięci niż jest faktycznie dostępne. Wtedy funkcja zawsze zwraca 1. Zauważmy, że powyższy wzór daje jedynie przybliżony wynik. Dlatego pojawia sie w nim dwuprocentowy margines błędu.

4.5 Funkcja `do_munmap`

Funkcja usuwa mapowanie wskazanego przedziału i zwraca 0 jeżeli operacja zakończyła się sukcesem. W przypadku porażki zwracany jest kod błędu.

```
int do_munmap(unsigned long addr, size_t len)
```

Algorytm 2 do_munmap

Dane wejściowe: `addr`, `len`

Wynik: Wynik

jeśli parametry są niepoprawne **to**
    porażka
koniec jeśli
jeśli nie istnieją w mapie pamięci obszary przecinające się z podanym zakresem **to**
    koniec
koniec jeśli

{O}dmapowywanie może wiązać się z koniecznością przecięcia obszaru mapowania na dwie części. Na zmiennej `extra` mamy zaalokowaną strukturę do zapamiętania informacji o dodatkowym obszarze, który powstanie w takiej sytuacji.

`extra` := nowa struktura `vm_area_struct`
`free` := lista obszarów w mapie pamięci przecinających się z podanym zakresem
dla wszystkich `mpnt` z `free` **wykonuj**
    jeśli jest zdefiniowana operacja `unmap` w `mpnt` **to**
        wykonaj `mpnt->vm_ops->unmap`
    koniec jeśli
    usuń mapowanie `mpnt`
    zmniejsz licznik mapowań
    zdezaktualizuj pamięć podręczną procesora
    napraw mapowanie (funkcja `unmap_fixup`)
koniec dla
zwolnij `extra` jeżeli nie zostało wykorzystane
zwolnij odmapowane strony w tablicy stron (funkcja `free_pgtables`)

Usunięcie mapowania polega na usunięciu struktur vm_area_struct opisujących obszary zawarte w obszarze odmapowywanym, oraz zmniejszenia obszarów przecinających się obszarem odmapowywanym.

Funkcja unmap_fixup otrzymuje jako parametr strukture vm_area_struct, którą ma być zlikwidowana lub zmniejszona, oraz dodatkową strukturę (zmienną extra). W zależności od sytuacji odpowiednio ją zmienia. Jeżeli konieczne jest utworzenie dodatkowej struktury, to wykorzystuje do tego zmienną extra.

4.6 Funkcja do_mmap

Funkcja do_mmap służy do stworzenia nowego mapowania pamięci. Można ją wykorzystać do zamapowania pliku, urządzenia, dzielonego lub "zwykłego" obszaru pamięci. Funkcja sys_brk wywołuje ją w ostatnim celu.

```
unsigned long do_mmap(struct file * file, unsigned long addr, unsigned long len,
unsigned long prot, unsigned long flags, unsigned long off)
```

W przedstawieniu algorytmu dla przejrzystości pominięto parametry określające sposób ochrony i inne atrybuty tworzonego mapowania.

5 Uwagi

5.1 Wolna pamięć i margines bezpieczeństwa

Parametry wzoru występujący w funkcji vm_enough_memory są przykładem wyniku uzyskanego drogą eksperymentu. Zmiana wartości buffer_mem.min_percent i buffer_mem.min_percent może spowodować znaczny spadek wydajności systemu. Interesującym doświadczeniem mogłaby być zmiana występującego tam dwuprocentowego marginesu bezpieczeństwa.

5.2 Slab allocator

Począwszy od wersji 2.2 do przydzielania pamięci na struktury jądra m.in. vm_area_struct wykorzystywany jest allokator pamięci **Slab** [Bonwick]. W odróżnieniu od **kmalloc** nie znajduje on za każdym razem fragmentu pamięci dla allokowanej struktury, tylko przeznacza pewną liczbę stron na struktury każdego typu. Pozwala to optymalizować przydział pamięci na struktury, które występują w największej liczbie egzemplarzy.

6 Pliki źródłowe

Omawiane funkcje zdefiniowane są w pliku mm/mmap.c. Deklaracje struktur znajdują się w plikach include/linux/shed.h i include/linux/mm.h

Algorytm 3 do_mmap

Dane wejściowe: `file`, `addr`, `len` oraz dodatkowe parametry opisujące strukturę `vm_area_struct`, która powstanie, nie wpływające w istotny sposób na algorytm.

Wynik: 0 lub kod błędu

```
jeśli parametry są niepoprawne to
   porażka
koniec jeśli
jeśli jest już zbyt wiele mapowań pamięci lub jest założona blokada na mapowanie to
   porażka
koniec jeśli
jeśli mapowanie pliku, do którego brakuje wystarczających uprawnień to
   porażka
koniec jeśli
jeśli nie podano konkretnego adresu obszaru mapowania to
   addr := wolny obszar (funkcja get_unmapped_area)
   jeśli wolny obszar nie istnieje to
      porażka
   koniec jeśli
koniec jeśli
vma := nowa struktura vm_area_struct
inicjuj vma (ustawienie adresów, flag dostępu etc.)
usuń stare mapowania obszaru
jeśli przekroczone zostaną limity lub brakuje pamięci to
   porażka
koniec jeśli
jeśli mapowanie pliku to
   zamapuj plik (funkcja file->f_op->mmap)
   jeśli wystąpił błąd to
      porażka
   koniec jeśli
koniec jeśli
dodaj vma do mapy pamięci (funkcja insert_vm_struct)
jeżeli możliwe, scal vma z sąsiednimi segmentami (funkcja merge_segments)
```

Bibliografia

[Rubini 99] Rubini A.: *Linux-sterowniki urządzeń* Warszawa, RM 1999.

[Rusling 99] Rusling D. A.:*The Linux kernel* Linux Documentation Project 1999

[Bonwick] Bonwick J.:*The Slab Allocator: An Object-Caching Kernel Memory Allocator* Sun Microsystems