Rozdział 1

Odwzorowywanie plików do pamięci

Autorzy: Radek Bartosiak i Patryk Czarnik

Jedną z najważniejszych cech systemu operacyjnego Linux jest fakt, że używa on pamięci wirtualnej. Mechanizm ten umożliwia utrzymywanie w przestrzeni adresowej procesu danych, które nie są aktualnie obecne w fizycznej pamięci operacyjnej. Dzieje się tak przede wszystkim w sytuacji, gdy wszystkie dane z wirtualnej przestrzeni adresowej nie mieszczą się w pamięci fizycznej – wówczas dane są tymczasowo zapisywane w pamięci pomocniczej (np. na dysku) i sprowadzane do pamięci fizycznej później (jest to **mechanizm wymiany** - *swapping*).

Możliwe jest jednak także odwzorowywanie w przestrzeni adresowej procesu danych, które już na początku są zapisane na zewnętrznym nośniku danych, a są sprowadzane do pamięci fizycznej dopiero w razie potrzeby odczytu lub modyfikacji. To właśnie realizuje mechanizm zwany **odwzorowywaniem plików do pamięci** (mapowaniem plików, stronicowaniem plików) - *file mapping*.

Czemu służy mapowanie plików do pamięci? Kod i dane większości programów (procesów) są przechowywane na urządzeniach zewnętrznych takich jak dysk czy CD-ROM. Pliki te bywają duże, w porównaniu z rozmiarem pamięci operacyjnej. Mapowanie pozwala sprowadzać do pamięci operacyjnej te fragmenty plików z kodem czy danymi, które są rzeczywiście potrzebne. Potrzebne są zasadniczo w dwóch przypadkach: pobrania informacji lub zmiany informacji zawartej na nośniku. W dodatku mechanizm mapowania sprawdza wówczas czy potrzebny fragment pliku jest już w pamięci operacyjnej – mógł zostać już wcześniej zmapowany. Takie postępowanie czyni procesy mniejszymi, dzięki czemu łatwiej mieszczą się w pamięci operacyjnej, rzadziej żądają zwolnienia fizycznych stron pamięci. Programy w systemie działają dzięki temu znacznie szybciej.

1.1 Pamięć wirtualna

Mapowanie plików służy realizacji jednego z kluczowych zadań systemu operacyjnego, jakim jest szybka i efektywna wymiana informacji (czyli danych - choć tu danymi może być kod programu) między procesorem i pamięcią pomocniczą (także sprzętem). Zasób jakim jest szybka fizyczna pamięć operacyjna jest drogi, ma małą pojemność, źle nadaje się do długoterminowego przechowywania danych. Powszechnie dane są przechowywane na nośnikach typu dyski, CD-ROMy itp. Efektywność wymiany polega na zminimalizowaniu wymagań sprzętowych, tak by implementacja mogła być powszechnie stosowana. Szybkość, w dobie interakcyjnego korzystania z komputerów, jest zaś najważniejszym wymaganiem stawianym przed każdym programem, szczególnie tak ważnym jak systemem operacyjny.

By ten cel osiągnąć w Linuxie, jak w dobrej drużynie, współdziałają ze sobą różne mechanizmy: stronicowanie i pamięć wirtualna (to jest podstawa) oraz mapowanie plików i **podręczna pamięć buforowa stron** - *page cache* (tzn. szybkie sprawdzanie czy strona jest w pamięci fizycznej i jej sprowadzanie) wraz z mechanizmem **kopiowania na żądanie** - *copy on write* i dynamicznym ładowaniem (tzn. pobieraniem stron naprawdę potrzebnych). Jest to dokładnie opisane w rozdziałach 30 - 34.

W Linuxie wiele procesów wykonuje się współbieżnie. Każdy proces może adresować pamięć za pomocą **adresu liniowego** (*linear address*) – liczby należącej do **liniowej przestrzeni adresowej** *linear address space*, tzn. przedziału [0,`TASK_SIZE`]. W tej przestrzeni wydzielone są spójne, nieprzecinające się **obszary pamięci wirtualnej** (*virtual memory areas*), w skrócie obszary VMA. System przechowuje dla każdego procesu katalog stron, pośrednie katalogi stron oraz tablice stron pozwalające na odnalezienie miejsca fizycznego zapisu danych (strony w pamięci fizycznej miejsca w pliku lub urządzeniu wymiany). System tworzy procesowi iluzję jedyności. Nie każdy adres z przestrzeni adresowej jest jednak poprawny. Cóż to byłby za świat bez "segmentation faults"? Adres jest niepoprawny jeśli proces nie ma obszaru pamięci wirtualnej (VMA), który go obejmuje.

1.2 Działanie mechanizmu mapowania

Mapując plik tworzymy obszar pamięci wirtualnej, w tablicy stron nie umieszczamy jednak wskaźników do odpowiednich stron w pamięci fizycznej. Jeśli proces będzie chciał korzystać z danych z tego obszaru VMA, przerwanie braku strony spowoduje (w uproszczeniu) sprowadzenie danych do pamięci fizycznej.

Mapowanie plików może odbywać się na dwa sposoby: **dzielony** i **prywatny**. Jeśli plik zmapowany jest w sposób dzielony, to modyfikacja w obszarze pamięci powoduje zmianę zawartości pliku. W przypadku mapowania w sposób prywatny zmiany nie są widoczne dla innych procesów ani nie wpływają na zawartość pliku. Można również utworzyć **anonimowy obszar pamięci wirtualnej** *anonymous virtual memory area*. Jest to obszar pamięci wirtualnej nie związany z żadnym plikiem. Obszary anonimowe mogą być tylko prywatne.

W przypadku mapowania w sposób dzielony zmiany dokonane w pamięci nie są od razu zapisywane do pliku na dysku – niezmiernie spowolniłoby to operacje na pamięci. Zmiany są zapisywane gdy wywoła się funkcję `vm_op->sync` (patrz "Struktury danych") lub przy odmapowaniu obszaru. Mechanizm *page cache* powoduje jednak, że wszystkie procesy czytając ten plik (przez mapowanie go, czytanie blokowe (`read`) jak i strumieniowe (np. `fgetc`)) przeczytają jego zmienioną zawartość z pamięci fizycznej. Tak więc z punktu widzenia użytkownika - programisty zmiany w obszarze pamięci wirtualnej mapującym plik w sposób dzielony pociągają za sobą od razu zmiany w pliku. Tylko w wyjątkowych sytuacjach (awaria sprzętu, przerwanie zasilania czy wyjęcie dyskietki z mapowanym plikiem) może się okazać, że fizycznie plik nie został zmieniony.

W przypadku mapowania w sposób prywatny stosowany jest mechanizm **kopiowania na żądanie** (*copy on write*). Gdy fragment pliku mapowany przez pewnien proces jest już wczytany i w fizycznej pamięci znajdują się strony z jego zawartością, nie są one kopiowane – proces może czytać bezpośrednio z nich. Jednak jeśli spróbuje coś w nich zapisać, wtedy zostaną one skopiowane, aby zmiany w pamięci nie były widoczne dla innych procesów.

Gdy tworzony jest anonimowy obszar pamięci wirtualnej, jest mu przypisywana **strona pusta** (*null page*) – strona cała wypełniona wartościami zerowymi. Proces może z niej czytać (ale nie ma tam nic ciekawego). Dopiero gdy proces zechce w tym obszarze coś zapisać, strona, do której się odwołuje, zostanie skopiowana i zmiany w tym obszarze będą widoczne tylko lokalnie dla jednego procesu. W systemie może więc istnieć tylko jedna strona pusta, a fizyczna pamięć dla procesów może być przydzielana tylko gdy jest naprawdę potrzebna. Jest to jedno z zastosowań mechanizmu *copy on write*.

1.3 Struktury danych

W celu zarządzania procesami jądro Linuxa utrzymuje wiele różnych informacji. Pokrótce przyjrzyjmy się tym strukturom jądra które związane są z mapowaniem. W Linuxie informacje o każdym z procesów przechowywane są w strukturze:

```
struct task_struct{   /* linux/shed.h */
  /* ...wiele innych informacji o procesie... */
  struct mm_struct *mm
  /* ...wiele innych informacji o procesie... */
}
```

Nas interesuje pamięć procesu. Dane o niej znajdują sie w strukturze wskazywanej przez wskaźnik `mm`, która określa i przechowuje informację o obszarach pamięci wirtualnej danego procesu.

Przedstawiamy tylko interesujące dla mechanizmu mapowania pola tej struktury.

```
struct mm_struct {         /* linux/mm.h */
         struct vm_area_struct *mmap;
         struct vm_area_struct *mmap_avl;
         struct vm_area_struct *mmap_cache;
          /* ... */
         int map_count;
         struct semaphore mmap_sem;
          /* ... */
         unsigned long def_flags;
          /* ... */
         };
```

Poszczególne pola mają następujące znaczenie:

- `mmap` – lista obszarów VMA procesu, posortowana według adresów (obszary się nie przecinają, nie ma więc znaczenia czy według początkowych czy końcowych);
- `mmap_avl` – drzewo AVL struktur VMA, kluczem jest też adres (drzewo jest incjowane tylko gdy liczba obszarów VMA jest nie mniejsza od `AVL_MIN_MAP_COUNT = 32`)
- `mmap_cache` – ostatni obszar VMA procesu znaleziony przez funkcję `find_vma`, chodzi o to by statystycznie usprawnić jej pracę;
- `map_count` – liczba obszarów VMA procesu;

- `mmap_sem` – semafor gwarantujący jedyność pisarza modyfikującego strukturę;
- `def_flags` – domyślne flagi, jakie mają być nakładane na obszary VMA procesu.

Każdy obszar pamięci wirtualnej jest opisywany przez strukturę `vm_area_struct` (strukturę VMA). Jest dokładnie jedna taka struktura dla każdego obszaru pamięci wirtualnej każdego procesu.

```
struct vm_area_struct {        /* linux/mm.h */
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;

        pgprot_t vm_page_prot;
        unsigned short vm_flags;

        short vm_avl_height;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;

        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        struct vm_operations_struct * vm_ops;
        unsigned long vm_offset;
        struct file * vm_file;
        unsigned long vm_pte;
};
```

- `vm_mm` – wskaźnik do struktury `mm_struct` procesu;
- `vm_start` – adres początku obszaru;
- `vm_end` – adres końca obszaru;
- `vm_next` – wskaźnik do następnej struktury na liście `vm_mm->mmap`;
- `vm_page_prot` – rzeczywiste prawa dostępu do pamięci z uwzględnieniem ograniczeń sprzętowych realizacji ochrony pamięci (patrz niżej);
- `vm_flags` – bitowa alternatywa różnych flag (patrz niżej)
- `vm_avl_height`,
- `vm_avl_left`,
- `vm_avl_right` – drzewo `vm_mm->mmap_avl`;
- `vm_next_share`,

Tablica 1.1: Sprzętowe ograniczenia ochrony pamięci dla procesora i386

	PROT_NONE	PROT_READ	PROT_WRITE	PROT_EXEC
MAP_SHARED	r: (no) no	r: (yes) yes	r: (no) yes	r: (no) yes
	w: (no) no	w: (no) no	w: (yes) yes	w: (no) no
	x: (no) no	x: (no) yes	x: (no) yes	x: (yes) yes
MAP_PRIVATE	r: (no) no	r: (yes) yes	r: (no) yes	r: (no) yes
	w: (no) no	w: (no) no	w: (copy) copy	w: (no) no
	x: (no) no	x: (no) yes	x: (no) yes	x: (yes) yes

- `vm_pprev_share` – lista cykliczna związana z inode'm (`vm_file->dentry->inode->i_mmap`) (jeśli w tym obszarze mapowany jest plik), lista odwzorowań dla obszarów pamięci dzielonej IPC lub nieużywane; ciekawym rozwiązaniem jest to, że `vm_pprev_share` jest wskaźnikiem bezpośrednio do pola `vm_next_share` w strukturze poprzednika;
- `vm_ops` – wskaźnik do zestawu operacji na pamięci, który jest zależny od rodzaju pamięci (prywatna czy dzielona) i potencjalnie od systemu plików; operacje te opisujemy dalej;
- `vm_offset` – jeśli mapowany jest plik - pozycja w pliku odpowiadająca początkowi obszaru; używane także przy pamięci dzielonej;
- `vm_file` – jeśli mapowany jest plik - wskaźnik do opisującej go struktury `file`;
- `vm_pte` – używane tylko przez mechanizm pamięci dzielonej IPC.

Ze względu na sprzętowe ograniczenia ochrony pamięci, w architekturze i386 pewnych kombinacji uprawnień nie da się zrealizować. Tablica przedstawia jakie uprawnienia realizowane sprzętowo odpowiadają uprawnieniom podanym przez użytkownika.

Flagi, które mogą być zaznaczone w `vm_flags`:

- `VM_READ, VM_WRITE, VM_EXEC` – aktualne prawa do odczytu, zapisu i wykonywania;
- `VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC` – dopuszczalne prawa do odcztu, zapisu i wykonywania danych zapisanych w obszarze;
- `VM_SHARED` – zaznaczone jeśli jest to obszar dzielony;
- `VM_MAYSHARED` – zaznaczone jeśli dla użytkowników obszar powinien być widoczny jako dzielony (nawet jeśli w rzeczywistosci został zredukowany do prywatnego – dzieje się tak przy mapowaniu dzielonym z prawami tylko do odczytu);
- `VM_GROWSDOWN` – zaznaczone, jeśli obszar ma strukturę stosu;
- `VM_GROWSUP` – zaznaczone, jeśli obszar ma strukturę odwrotną do stosu;
- `VM_SHM` – zaznaczone, jeśli obszar jest obszarem pamięci dzielonej IPC, zapobiega wymianie na urządzenie wymiany;

- `VM_DENYWRITE` – oznacza, że mapowany plik może być otwarty do pisania tylko przez jeden proces;
- `VM_EXECUTABLE` – oznacza, że to obszar z kodem wykonywalnym;
- `VM_LOCKED` – strony tego obszaru będą na stałe umieszczone w pamięci fizycznej;
- `VM_IO` – obszar mapuje urządzenie wejścia/wyjścia, a nie plik dyskowy;

Zestaw operacji na obszarach pamięci wirtualnej, który może być różny dla różnych typów tych obszarów (jest inny dla prywatnego mapowania plików, inny dla dzielonego, jeszcze inny dla obszarów pamięci dzielonej IPC) jest określony w strukturze:

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    void (*unmap)(struct vm_area_struct *area, unsigned long, size_t);
    void (*protect)(struct vm_area_struct *area, unsigned long, size_t, unsigned int newprot);
    int (*sync)(struct vm_area_struct *area, unsigned long, size_t, unsigned int flags);
    void (*advise)(struct vm_area_struct *area, unsigned long, size_t, unsigned int advise);
    unsigned long (*nopage)(struct vm_area_struct * area, unsigned long address, int write_access);
    unsigned long (*wppage)(struct vm_area_struct * area, unsigned long address,
             unsigned long page);
    int (*swapout)(struct vm_area_struct *, struct page *);
    pte_t (*swapin)(struct vm_area_struct *, unsigned long, unsigned long);
};
```

Nie wszystkie z tych operacji muszą być określone – w niektórych polach struktury mogą być wartości NULL. Jeśli jednak są określone (przez wskaźniki do właściwych funkcji), są wykonywane w następujących okolicznościach:

- `open` – przy otwarciu obszaru VMA, które następuje m.in. przy tworzeniu obszaru, gdy fork kopiuje obszary istniejącego procesu do nowego;
- `close` – przy likwidowaniu obszaru;
- `unmap` – przy odłączaniu obszaru;
- `protect` – służy do zakładania nowych znaczników ochrony;
- `sync` – przy mapowaniu plików w sposób dzielony zapisuje do pliku zmiany;
- `advise` – obecnie nie używana;
- `nopage` – wywoływana przy błędzie braku strony;
- `wppage` – wywoływana przy próbie zapisu do obszaru chronionego przed zapisem;
- `swapout` – używana przy wymianie stron z pamięci na dysk; zwraca 1 jeżeli operacja powiodła, 0 w przypadku błędu;

- `swapin` – używana przy wymianie stron z dysku do pamięci; zwraca przydzieloną ramkę;

Operacje te są określane przez specyficzną dla systemu plików operację mapującą (`f_op->mmap>`), więc potencjalnie mogą tu być wstawione niestandardowe funkcje. Przy normalnym mapowaniu plików funkcja `generic_filemmap` wstawia tutaj funkcje, które opisujemy w podrozdziale “Funkcje charakterystyczne dla typu obszaru i systemu plików”.

1.4 Najważniejsze funkcje związane z mapowaniem

W tym podrozdziale opisujemy działanie dwóch podstawowych funkcji systemowych związanych z mapowaniem plików: funkcji mapującej `do_mmap()` i odmapowującej `do_munmap`.

1.4.1 Opis funkcji do_mmap()

Zadaniem funkcji `do_mmap` jest utworzenie nowego obszaru pamięci wirtualnej, czyli w praktyce stworzenie nowej struktury `vm_area_struct` , zainicjowanie jej pól odpowiednimi wartościami i wstawienie do struktur opisujących pamięć wirtualną procesu.

Funkcja jest zdefiniowana w pliku mm/mmap.c, oto jej nagłówek:

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

Parametry mają następujące znaczenie:

- `file` – określenie pliku, który jest mapowany, lub NULL, jeśli ma zostać stworzony anonimowy obszar pamięci wirtualnej. Plik powinien być otwarty z odpowiednimi prawami (patrz `prot`).

- `addr` – adres w pamięci wirtualnej, pod którym ma być początek tworzonego obszaru, lub NULL. Jeśli jest to możliwe bez odmapowywania żadnych obszarów, funkcja utworzy obszar pod tym adresem, jeśli nie - pod innym, chyba że podano opcję `MAP_FIXED` (patrz `flags`).

- `len` – długość obszaru;

- `prot` – prawa odczytu/zapisu/wykonywania na tworzonym obszarze podane jako bitowa alternatywa wybranych wartości z `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` lub wartość `PROT_NONE` oznaczająca brak jakichkolwiek uprawnień dostępu (trochę niepraktyczne). Jeśli mapowany jest plik, powinien być otwarty z prawami nie mniejszymi niż podane w `prot`, przy czym otwarcie do pisania umożliwia też wykonywanie.

- `flags` – dodatkowe flagi specyficzne dla obszarów pamięci wirtualnej. Może to być jedna z wartości `MAP_PRIVATE, MAP_SHARED`, które oznaczają czy ma być tworzony obszar dzielony, czy prywatny. Jeśli nie jest mapowany plik, musi to być `MAP_PRIVATE`. Dodatkowo (alternatywa bitowa) może być podana flaga `MAP_FIXED`, która powoduje, że funkcja umieszcza obszar pod adresem `addr` nawet jeśli wymaga to odmapowania umieszczonych tam już obszarów. Jeśli nie da się umieścić obszaru pod podanym adresem, funkcja kończy się niepowodzeniem.

- `off` – jeśli mapowany jest plik, pozycja w pliku, która będzie odpowiadać początkowi obszaru pamięci wirtualnej. Jeśli funkcją mapującą specyficzną dla pliku jest `generic_filemmap` (a jest tak dla normalnych plików dyskowych) i plik mapowany jest w sposób dzielony, wartość ta powinna być wielokrotnością rozmiaru strony.

Funkcja zwraca adres początku utworzonego obszaru lub ujemny kod błędu systemowego.

Działanie funkcji opiszemy komentując jej kod.

```
unsigned long do_mmap(struct file * file, unsigned long addr, unsigned long len,
     unsigned long prot, unsigned long flags, unsigned long off)
{
/* Zmienna mm jest ustawiana na strukturę pamięci procesu wołającego do_mmap */
     struct mm_struct * mm = current->mm;
     struct vm_area_struct * vma;
     int error;

/* Jeśli mapowany jest plik, musi udostępniać operację mapowania specyficzną dla systemu plików */
     if (file && (!file->f_op || !file->f_op->mmap))
             return -ENODEV;

/* Wielkość mapowanego obszaru wyrównywana jest do wielokrotności rozmiaru strony. */
/* Żądanie obszaru o długości 0 może być spełnione od razu. */
     if ((len = PAGE_ALIGN(len)) == 0)
             return addr;

/* Sprawdzenie czy nowy obszar zmieści się w maksymalnym rozmiarze procesu. */
     if (len > TASK_SIZE || addr > TASK_SIZE-len)
             return -EINVAL;


/* Sprawdzenie czy nie zostanie przekroczony zakres typu używanego do adresowania */
/* unsigned long int */
     if (off + len - 1 < off)
         return -EINVAL;

/* Czy proces nie przekroczy swojego limitu mapowań. */
     if (mm->map_count > MAX_MAP_COUNT)
             return -ENOMEM;

/* Jeżeli domyślnie mapowany obszar ma być na stałe umieszczony w pamięci fizycznej... */
     if (mm->def_flags & VM_LOCKED) {
             unsigned long locked = mm->locked_vm << PAGE_SHIFT;
             locked += len;
             if (locked < len)
                     return -EAGAIN;
/* Czy proces nie przekroczy swojego limitu zablokowanej pamieci ? */
             if ((current->rlim[RLIMIT_MEMLOCK].rlim_cur < RLIM_INFINITY) &&
```

```
            (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur))
                return -EAGAIN;
    }

/* Proste sprawdzenie praw dostepu do otwartego pliku */
    if (file != NULL) {
            switch (flags & MAP_TYPE) {
            case MAP_SHARED:
/* Proces musi mieć plik otwarty do pisania,by móc go zmapować z prawami zapisu */
                    if ((prot & PROT_WRITE) && !(file->f_mode & 2))
                            return -EACCES;

/* Nie można pozwolić na modyfikację pliku "tylko do dopisywania" */
                    if (IS_APPEND(file->f_dentry->d_inode) && (file->f_mode & 2))
                            return -EACCES;
/* Jeśli założona jest blokada obowiązkowa na plik, nie można go zmapować */
                    if (locks_verify_locked(file->f_dentry->d_inode))
                            return -EAGAIN;

                    /* fall through */
            case MAP_PRIVATE:
/* Zawsze potrzebne jest prawo do czytania */
                    if (!(file->f_mode & 1))
                            return -EACCES;
                    break;

            default:
                return -EINVAL;
            }
    }
/* Tylko pliki mogą być mapowane w sposób dzielony */
      else if ((flags & MAP_TYPE) != MAP_PRIVATE)
            return -EINVAL;

/* Gdy proces żąda przydzielenia konkretnego adresu,
 * musi on być adresem początku strony */
    if (flags & MAP_FIXED) {
            if (addr & ~PAGE_MASK)
                    return -EINVAL;
    } else {
/* get_unmapped_area zwraca adres, taki że obszar
 * [addr,addr+len] jest niezamapowany */
            addr = get_unmapped_area(addr, len);
            if (!addr)
                    return -ENOMEM;
    }
```

```
/* Alokowanie pamieci na strukturę vma_area_struct mapowanego obszaru */
     vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
     if (!vma)
             return -ENOMEM;
/* Incjowanie wartości */
     vma->vm_mm = mm;
     vma->vm_start = addr;
     vma->vm_end = addr + len;

/* We flags również są utrzymywane informacje o prawach dostępu,
 * na razie tylko we flagach bez "MAY";
 * dodanie do flag tych domyślnych z mm */
     vma->vm_flags = vm_flags(prot,flags) | mm->def_flags;

/* Operacje na flagach MAY... To maksymalne uprawnienia jakie może
 * dostać proces dla danego pliku i typu mapowania. */

     if (file) {
/* Najpierw dajemy prawa do wszystkiego, przy mapowaniu prywatnym i tak
 * zmiany nie zapiszą się do pliku */
             if (file->f_mode & 1)
                     vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;

/* Przy mapowaniu dzielonym... */
             if (flags & MAP_SHARED) {
                     vma->vm_flags |= VM_SHARED | VM_MAYSHARE;

/* ...jeśli plik nie jest otwarty do czytania, można zamiast mapowania
 * dzielonego przeprowadzić mapowanie prywatne.
 * Flaga VM_MAYSHARE zapewni, że /proc/xxx/maps będzie wypisywał ten obszar
 * jako dzielony. */
                     if (!(file->f_mode & 2))
                             vma->vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
             }
     } else
/* Do obszarów anonimowych są wszystkie prawa dostępu */
             vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;

/* Zmienione prawa zapamiętywane są w prot, ale z uwzględnieniem
 * sprzętowych ograniczeń realizacji ochrony pamięci */
     vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];

/* Dalsze inicjowanie pól struktury, niektóre puste i zerowe wartości
 * zostaną zastąpione dalej. */
     vma->vm_ops = NULL;
     vma->vm_offset = off;
```

```
     vma->vm_file = NULL;
     vma->vm_pte = 0;

/* Odmapowanie obszaru, będzie potrzebne tylko gdy MAP_FIXED  */
     error = -ENOMEM;
     if (do_munmap(addr, len))
             goto free_vma;

/* Sprawdzenie czy nie zostanie przekroczony zakres typu używanego do adresowania */
/* unsigned long int */

     if ((mm->total_vm << PAGE_SHIFT) + len < len)
             goto free_vma;

/* Czy proces nie przekroczy swojego limitu wirtualnej pamieci ? */
     if ((current->rlim[RLIMIT_AS].rlim_cur < RLIM_INFINITY) &&
         ((mm->total_vm << PAGE_SHIFT) + len
         > current->rlim[RLIMIT_AS].rlim_cur))
             goto free_vma;

/* Obszaru zmapowanego prywatnie do pisania nie można będzie wyrzucić z pamięci
 * do pliku, więc trzeba sprawdzić czy zmieści się w pamięci fizycznej i
 * urządzeniach wymiany (swapach) oraz buforach. */
     if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
         !(flags & MAP_NORESERVE)                             &&
         !vm_enough_memory(len >> PAGE_SHIFT))
             goto free_vma;

     if (file) {
/* Jeśli mapowany jest plik, powinien mieć określoną operację mmap,
 * którą teraz się wywołuje. M.in. wypełnia ona pole vm_ops. */
             int correct_wcount = 0;
             if (vma->vm_flags & VM_DENYWRITE) {
/* Przy tej fladze ustawionej nie można mapować jeśli
 * ktoś już może zapisywać do pliku. */
                     if (file->f_dentry->d_inode->i_writecount > 0) {
                             error = -ETXTBSY;
                             goto free_vma;
                     }
/* Ta operacja na i_writecount zabezpiecza przed otwarciem pliku do
 * pisania przez inny proces w razie zaśnięcia
 * w funkcji mmap. (generic_file_mmap nie zasypia, ale podmienione
 * funkcje mogą.) */
                     file->f_dentry->d_inode->i_writecount--;
                     correct_wcount = 1;
             }
```

```
                error = file->f_op->mmap(file, vma);
                if (correct_wcount)
                        file->f_dentry->d_inode->i_writecount++;
                if (error)
                        goto unmap_and_free_vma;
                vma->vm_file = file;
                file->f_count++;
        }

        flags = vma->vm_flags;
        addr = vma->vm_start;

/* Wstawienie struktury do list i drzewa. */
        insert_vm_struct(mm, vma);
/* Jeśli to możliwe, nowy obszar zostanie połączony z istniejącymi. */
        merge_segments(mm, vma->vm_start, vma->vm_end);

        mm->total_vm += len >> PAGE_SHIFT;
        if (flags & VM_LOCKED) {
/* Jeśli obszar ma być na stałe umieszczony w pamięci fizycznej, trzeba
 * zablokować strony, które obejmuje. */
                mm->locked_vm += len >> PAGE_SHIFT;
                make_pages_present(addr, addr + len);
        }

/* Sukces */
        return addr;

/* W przypadku błędu trzeba cofnąć dokonane dotąd zmiany. */
unmap_and_free_vma:
        flush_cache_range(mm, vma->vm_start, vma->vm_end);
        zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
        flush_tlb_range(mm, vma->vm_start, vma->vm_end);
free_vma:
        kmem_cache_free(vm_area_cachep, vma);
        return error;
}
```

1.4.2 Opis funkcji do_munmap

Funkcja `do_munmap` jest zdefiniowana w pliku mm/mmap.c, oto jej nagłówek:

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

Funkcja odmapowuje te obszary pamięci wirtualnej, które znajdują się między `addr` a `addr+len`.

Działanie funkcji opiszemy komentując jej kod.

```
int do_munmap(unsigned long addr, size_t len)
{
     struct mm_struct * mm;
     struct vm_area_struct *mpnt, *prev, **npp, *free, *extra;

/* Adres  ma wskazywać dokladnie na początek strony.
/* Adresy w odmapowywanym obszarze mają być mniejsze od górnego ograniczenia
 * przestrzeni adresowej procesów - stałej TASK_SIZE */
     if ((addr & ~PAGE_MASK) || addr > TASK_SIZE || len > TASK_SIZE-addr)
             return -EINVAL;

/* Długość odmapowywanego przedziału będzie zaokrąglona w górę do najbliższej
 * wielokrotności długości strony */
     if ((len = PAGE_ALIGN(len)) == 0)
             return -EINVAL;

/* Znajdujemy obszary (reprezentowane przez struktury vm_area_struct)
 * takie, że przecianają się one z odmapowywanym przedziałem. */
     mm = current->mm;

/* Na mpnt wpisywany jest pierwszy w kolejności adresów, dotknięty przez
 * odmapowanie obszar przecinający odmapowywany przedział. */
     mpnt = find_vma_prev(mm, addr, &prev);
     if (!mpnt)
             return 0; /* nie ma niczego do odmapowania. */

/* Teraz mpnt jest najmniejszym obszarem takim że, addr < mpnt->vm_end */
     if (mpnt->vm_start >= addr+len)
             return 0; /* nie ma niczego do odmapowania */

/* Jeśli odmapowujemy fragment spójnego dotychczas obszaru
 * (tzn. reprezentowanego w jednej strukturze vm_area_struct),
 * powstanie luka i dwa mniejsze obszary VMA,
 * będziemy potrzebować dodatkowej struktury vm_area_struct.
 * Musimy sprawdzić czy nie przekroczymy limitu liczby struktur VMA,
 * oczywiście to jest sprawdzanie na zapas. I alokowanie na zapas.
 * Bo dodatkowa struktura jest potrzebna tylko w jednym specyficznym  przypadku. */
     if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
         && mm->map_count >= MAX_MAP_COUNT)
             return -ENOMEM;

/* ... a tak może nie być na tę strukturę miejsca w pamięci */
     extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
     if (!extra)
             return -ENOMEM;
```

```
/* nnp (vm_area_struct **) wskazuje na wskaźnik do następnego
 * elementu listy obszarów VMA uporządkownej według adresów
 * npp to wskaźnik ustawiany tak, że zawsze wskazuje na mpnt */
     npp = (prev ? &prev->vm_next : &mm->mmap);
/* free to lista obszarów (reprezentowanych przez vm_area_struct),
 * które są dotknięte przez odmapowanie (przeznaczone do zwolnienia) */
     free = NULL;
/* Przesuwamy się po tej liście dopóki się nie skończy albo
 * nie wyjdziemy poza odmapowywany obszar
     for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
             *npp = mpnt->vm_next;
/* Wstawianie do listy free elementu mpnt (jednoczesne usuwanie z listy ...) */
             mpnt->vm_next = free;
             free = mpnt;
/* Usuwanie zn drzewa avl */
             if (mm->mmap_avl)
                     avl_remove(mpnt, &mm->mmap_avl);
     }

/* Na liście 'free' znajdują sie obszary, które trzeba zwolnić */
     while ((mpnt = free) != NULL) {
             unsigned long st, end, size;
             free = free->vm_next;
             st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
             end = addr+len;
             end = end > mpnt->vm_end ? mpnt->vm_end : end;
             size = end - st;
/* Zwalniamy właściwą dla rodzaju zmapowanego w pamięci wirtualnej obszaru
 * funkcją */
             if (mpnt->vm_ops && mpnt->vm_ops->unmap)
                     mpnt->vm_ops->unmap(mpnt, st, size);
/* Jeśli zwalniany obszar był mapowaniami pewnego inode'a,
 * usuwamy ten obszar ze związanej z tym inode'm listy obszarów VMA.
 * Jeśli zmapowano z flagą VM_DENYWRITE zwiększamy licznik pisarzy
 * właściwego inoda o 1 */
             remove_shared_vm_struct(mpnt);
/* Zmniejszamy ilość mapowań związanych z procesem */
             mm->map_count--;
/* Zwalnianie zasobów związanych z obsługą pamięci przeznaczonej
 * na odmapowywany obszar (elementy tablic stron, katalogów, etc.) mapowaniem */
             flush_cache_range(mm, st, end);
             zap_page_range(mm, st, size);
             flush_tlb_range(mm, st, end);

/* Uporządkownie (ewentualnie) struktur w jądrze związanych z mapowaniem  */
             extra = unmap_fixup(mpnt, st, size, extra);
```

```
    }

    if (extra)
/* Jeśli nie potrzeba dodatkowej struktury, zwalniamy pamięć */
            kmem_cache_free(vm_area_cachep, extra);
/* zwolnienie zasobów */
    free_pgtables(mm, prev, addr, addr+len);

    mm->mmap_cache = NULL; /* bo teraz mogą tu być śmieci */
    return 0;
}
```

1.4.3 Funkcje dostępne dla użytkownika

Dwie opisane powyżej funkcje oraz funkcja synchronizująca zawartość pliku są dostępne dla użytkownika - programisty jako funkcje systemowe w bibliotece języka C.

Funkcje te, odpowiednio obudowane w realizację wywołania systemowego i przeliczanie adresów, występują zadeklarowane w pliku sys/mman.h jako:

`void*` **mmap**`(void *addr, size_t len, int prot, int flags, int fd, off_t offset)`
`int` **munmap**`(void *addr, size_t len)`
`int` **msync**`(const void *addr, size_t length, int flags)`

Parametry mają znaczenie i dopuszczalne wartości takie jak dla wewnętrznych funkcji systemowych, z tą różnicą, że adresy (w parametrach i zwracany) są podane jako wskaźniki C, a nie jako adresy liniowe.

Pierwsza funkcja zwraca adres, pod którym został zmapowany nowy obszar, a w przypadku błędu `MAP_FAILED`. Druga i trzecia 0, a przypadku błędu -1. Wszystkie w razie błędu ustawiają zmienną `errno`.

Więcej informacji: man mmap, man msync.

1.5 Opis pozostałych funkcji z pliku mmap.c

Poniżej opisujemy pomocnicze funkcje poprawiające struktury danych związane z pamięcią wirtualną lub wyszukujące coś w tych strukturach. Operują one na strukturach procesu bieżącego lub procesu, którego struktura `mm_struct` jest parametrem funkcji.

`struct vm_area_struct *` **find_vma**`(struct mm_struct * mm, unsigned long addr)`
Znajduje strukturę VMA procesu opisującą pierwszy z lewej obszar, którego koniec znajduje się na prawo od parametru `addr` (`addr < vm_end`). Funkcja zwraca wskaźnik do znalezionej struktury lub 0 jeśli takiej nie ma. W szczególności jeśli `addr` znajduje się na zmapowanym obszarze pamięci wirtualnej, wynikiem będzie struktura VMA opisująca ten obszar.

Najpierw sprawdzana jest struktura VMA, do której wskaźnik zapamiętany jest w polu `mm->mmap_cache`; jeśli `addr` znajduje się wewnątrz obszaru opisywanego przez tę strukturę, to ona jest wynikiem.

Jeśli w `mm->mmap_cache` nie było szukanej struktury, znajdowana jest ona w drzewie AVL lub liście, jeśli nie ma drzewa. Wskaźnik do znalezionej struktury zapamiętywany jest w `mm->mmap_cache` i zwracany jako wynik funkcji.

`struct vm_area_struct *`**find_vma_prev**`(struct mm_struct * mm, unsigned long addr, struct vm_area_struct **pprev)`
Ta funkcja robi dokładnie to samo co poprzednia, ale dodatkowo pod adres `pprev` zapisuje wskaźnik do poprzedniej struktury VMA lub `NULL` jeśli takiej nie ma (gdy wynik jest pierwszą strukturą VMA).

`unsigned long` **get_unmapped_area**`(unsigned long addr, unsigned long len)`
Ta funkcja w przestrzeni adresowej bieżącego procesu znajduje niezamapowany obszar o długości `len`. Jeśli `addr != NULL`, szukanie rozpoczyna się od tego adresu i jeśli jest to początek takiego obszaru, właśnie `addr` zostanie zwrócony.

Zwracany jest adres początku szukanego obszaru lub 0 w przypadku błędu (który powinien być zinterpretowany jako błąd braku pamięci).

Funkcja używana jest przez funkcję `do_mmap` wywołaną bez ustawionej flagi `MAP_FIXED` oraz przez funkcję `shmat` z parametrem `NULL`.

Algorytm 1 get_unmapped_area

Dane wejściowe: addr, len
Wynik: niezamapowany fragment przestrzeni adresowej o długości len

jeśli addr == NULL **to**
addr := `TASK_UNMAPPED_BASE`
koniec jeśli
addr jest zaokrąglany w dół do początku strony
`vma := find_vma` – struktura VMA pierwszego zmapowanego obszaru, którego koniec jest na prawo od startowego adresu.
jeśli obszar mapowany przez vma rozpoczyna się dalej niż addr + len **to**
RETURN addr
w przeciwnym przypadku
dopóki nie koniec listy struktur vma **wykonuj**
przeglądaj listę
jeśli początek bieżącego - koniec poprzedniego >= len **to**
RETURN początek bieżącego
koniec jeśli
koniec dopóki
jeśli koniec ostatniego + len < TASK_SIZE **to**
RETURN koniec ostatniego
w przeciwnym przypadku
RETURN 0 (błąd braku pamięci)
koniec jeśli
koniec jeśli

`void` **merge_segments**`(struct mm_struct * mm, unsigned long start_addr, unsigned long end_addr)`
Jeśli wśród obszarów pamięci wirtualnej chociaż w części należących do zakresu [`start_addr`, `end_addr`] (i pierwszego obszaru na lewo od nich - ważne dla `do_mmap`) są takie, które można ze sobą połączyć, to ta funkcja to robi. “Łączenie” polega na zastąpieniu dwóch (lub więcej) struktur

VMA jedną.

Łączone są obszary, które spełniają następujące warunki:

- sąsiadują bezpośrednio ze sobą (`vm_end` jednego jest równy `vm_start` drugiego);
- mapują ten sam plik lub oba nie mapują pliku;
- ich struktury VMA mają te same wartości pól `vm_flags, vm_ops, vm_pte`;
- jeśli obszary mapują plik lub są to obszary pamięci dzielonej IPC dodatkowym warunkiem jest zachowanie ciągłości w adresowaniu pliku lub obszaru pamięci dzielonej ($offsetlewego + rozmiarlewego = offsetprawego$).

Funkcję należy wywoływać gdy opuszczony jest semafor dostępu do struktury `mm_struct`.

Algorytm 2 merge_segments

Dane wejściowe: adres początkowy, adres końcowy
Wynik:

Funkcją `find_vma_prev` znajdź strukturę VMA pierwszego obszaru z podanego wyżej zakresu (vma) i jego poprzednika (pop)
jeśli vma == NULL **to**
  RETURN //nie ma w ogóle zmapowanych obszarów w tym zakresie adresów
koniec jeśli
dopóki nie koniec listy struktur VMA i nie przekroczony adres końcowy **wykonuj**
  jeśli spełnione warunki złączenia vma z pop **to**
    rozszerz obszar pop do końca vma
    usuń strukturę VMA pop z listy i drzewa AVL
    zmniejsz o 1 `mm->mapcount`
    jeśli określona funkcja vm_ops->close **to**
      vma->vm_offset := vma->vm_offset + vma->vm_end - vma->vm_start
      vma->vm_start := vma->vm_end
      wywołaj vm_ops->close
    koniec jeśli
    `remove_shared_vm_struct` usuwa strukturę vma z listy cyklicznej związanej w plikiem lub obszarem pamięci dzielonej
    zwalniana jest pamięć używana przez usuwaną strukturę vma
  koniec jeśli
  pop := vma
  vma := vma->next
koniec dopóki
mm->mmap_cache := NULL // bo teraz mogą tam być śmieci

`static inline void` **remove_shared_vm_struct**`(struct vm_area_struct *vma)`
Jeśli `vma` opisuje obszar mapujący plik, usuwa podaną strukturę z cyklicznej listy sytruktur VMA związanej z inode'm (pole `i_mmap` w strukturze `inode`).

`void` **insert_vm_struct**`(struct mm_struct *mm, struct vm_area_struct *vmp)`

Dodaje nową strukturę VMA wskazywaną przez `vmp` do struktur opisujących pamięć wirtualną. Wstawia ją na odpowiednią pozycję do listy `mm->mmap`, do drzewa `mm->mmap_avl`, jeśli jest ono określone, i jeśli struktura opisuje obszar mapujący plik, także do listy cyklicznej związanej z inode'm.

`static void` **free_pgtables**`(struct mm_struct * mm, struct vm_area_struct *prev, unsigned long start, unsigned long end)`
Zwalnia niepotrzebne wpisy katalogu stron z zakresu adrsów od `start` do `end`. W szybszej pracy może pomóc parametr `prev`, który powinien wskazywać na strukturę VMA opisującą obszar przed podanym zakresem. Przy wywołaniu funkcji po usunięciu jakiejś struktury VMA, `prev` powinien wskazywać na jej poprzednika. Ten parametr może także być równy `NULL`, wówczas funkcja zadziała mniej efektywnie.

Podane adresy są wyrównywane do granicy stron, następnie przeglądana jest lista struktur VMA poczynając od `prev` lub początku listy, gdy `prev == NULL`. Gdy zostanie znaleziona pierwsza luka między zmapowanymi obszarami pamięci wirtualnej, choć częściowo zawarta w podanym zakresie adresów, dla tej luki i struktury `mm` zostanie wywołana funkcja `clear_page_tables` i już w tym momencie `free_pgtables` zakończy działanie.

Jak widać, funkcja nie musi zwolnić wszystkich wpisów w podanym zakresie, ale nadaje się do poprawiania katalogu stron po usunięciu jednej albo wszystkich struktur VMA.

`void` **exit_mmap**`(struct mm_struct * mm)`
Ta funkcja odmapowuje wszystkie obszary pamięci wirtualnej jednego procesu.

Najpierw na wskaźniki do listy i drzewa AVL w strukturze `mm` wstawiane są wartości `NULL` i zerowane są liczniki `rss, total_vm, locked_vm`. Następnie dla każdej struktury VMA wywoływane są operacje `vm_ops->unmap` i `vm_ops->close` (o ile są określone), zmniejszany jest licznik `mm->map_count`, wywoływana jest funkcja `remove_shared_vm_struct`, która usuwa strukturę z listy związanej z inode'm lub obszarem pamięci dzielonej, oraz funkcja `zap_page_range`. Zwalniana jest pamięć używana przez strukturę VMA. Po usunięciu wszystkich struktur VMA wywoływana jest funkcja `clear_page_tables`, która czyści wpisy w katalogu stron.

`static struct vm_area_struct *` **unmap_fixup**`(struct vm_area_struct *area, unsigned long addr, size_t len, struct vm_area_struct *extra)`
Funkcja przywraca poprawność struktur danych związanych z mapowaniem obszaru `*area`. Parametry wywołania to: `area` wskaźnik do struktury `vm_area_struct` obszaru, którego część zaczynająca się pod adresem `addr` i o długości `len` zostanie odmapowana, `extra` to wskaźnik na dodatkową strukturę `vm_area_struct`, potrzebną gdy po odmapowaniu powstaną dwa obszary.

Funkcja ustala, jaka część obszaru `*area` została odmapowana i zgodnie z tym koryguje informacje go dotyczące. `vm_area_struct` obszaru *area jest już usunięta z listy zmapowanych przez proces obszarów. Nowe struktury `vm_area_struct` będą musiała być wstawiona do tej listy gdy nie cały obszar `*area` ma być odmapowany (przypadki 2, 3 i 4). Cztery możliwe przypadki to:

1. Cały obszar został odmapowany.
2. Odmapowano początek obszaru.
3. Odmapowano koniec obszaru.
4. Odmapowano mniejszy obszar zawarty wewnątrz początkowego obszaru, tworząc dziurę w początkowym obszarze.

W przypadku 1) zwalnia się pamięć zajętą przez strukturę `vm_area_struct` odmapowywanego obszaru. W przypadkach 2) i 3) struktura ta jest wykorzystywana do zapamiętania informacji o pozostałym obszarze pamięci – to jest o końcu (przypadek 2) lub o początku (przypadek 3) pierwotnego obszaru. W przypadku 4) z jednego obszaru powstają dwa: koniec i początek, potrzeba więc dwóch struktur VMA. Tą drugą jest struktura wskazywana przez zmienną `extra`.

1.6 Operacje charakterystyczne dla typu obszaru i systemu plików

Podczas mapowania pliku wywoływana jest funkcja, na którą w strukturze pliku wskazuje wskaźnik `f_op->mmap`. Powinna ona wykonać operacje związane z mapowaniem charakterystyczne dla danego systemu plików i wstawić na pole `vm_ops` struktury VMA zestaw operacji (patrz niżej i "Struktury danych").

Domyślną w Linuksie funkcją, która jest wykonywana podczas mapowania normalnych plików dyskowych, jest zdefiniowana w pliku mm/filemap.c funkcja

`int` **generic_file_mmap**`(struct file * file, struct vm_area_struct * vma)`

Jeśli plik mapowany jest w sposób dzielony z prawami do zapisu, sprawdzane jest czy podany offset jest wielokrotnością rozmiaru strony. Jeśli nie, jest to błąd. Następnie funkcja wpisuje w pole `vm_ops` struktury `vma` zestaw operacji domyślnych dla mapowania dzielonego (patrz niżej). Dla mapowania prywatnego jako błąd traktowana jest sytuacja, gdy mapowany inode ma określoną operację `i_op->bmap` a offset nie jest wielokrotnością rozmiaru bloku dla danego inode'a. Jeśli nie ma takiego błędu, do `vm_ops` wpisywane są domyślne operacje dla obszarów prywatnych (patrz niżej). Błąd jest także zwracany w przypadku gdy inode nie ma określonej operacji `i_op->readpage`. Jeśli wszystko przebiegło pomyślnie, uaktualniany jest czas dostępu do inode'a i funkcja kończy się sukcesem.

Dla każdej struktury VMA w jej polu `vm_ops` określone są operacje na obszarze pamięci wirtualnej, które mogą być różne dla różnych typów obszarów i dla różnych systemów plików, które te obszary mapują. Poniżej opisujemy uniwersalne funkcje, które są przypisywane przez funkcję `generic_file_mmap`. Właśnie one są używane w przypadku normalnego mapowania normalnych plików dyskowych.

W przypadku mapowania plików w sposób prywatny określona jest jedynie operacja `nopage`. Wskaźnik wskazuje na funkcję `filemap_nopage`, która jest zdefiniowana w pliku mm/filemap.c.

`static unsigned long` **filemap_nopage**`(struct vm_area_struct * area, unsigned long address, int no_share)`

`area` to odpowiednia struktura VMA, `address` – adres przy którym nastapił błąd braku strony, `no_share` określa czy można jako wynik zwrócić adres strony, która już istnieje w pamięci i korzysta z niej inny proces, czy też wymagane jeat aby obecny proces miał własną kopię strony.

Funkcja korzysta w istotny sposób z mechanizmu *page cache*. W cache page umieszczane są strony które już znajdują się w pamięci opercyjnej lub zostały do niej sprowadzone niejako przy okazji, gdy istnieje duże prawdopodobieństwo że będą potrzebne.

Algorytm działania funkcji:

W przypadku mapowania w sposób dzielony określone (jako funkcje z pliku filemap.c) są operacje:

- `unmap` jako `filemap_unmap`,

Algorytm 3 filemap_nopage

Dane wejściowe: area, address, no_share
Wynik: wczytana strona pamięci

```
sprawdź czy strona jest w page cache
jeśli strona jest obecna w page cache to
  jeśli no_share to
    alokuj pamięć na kopię
    kopiuj stronę
  koniec jeśli
  sprowadź stronę do pamięci (w razie gdyby była wyswapowana)
  RETURN strona
w przeciwnym przypadku
  spróbuj wczytać stronę i następne strony (try_to_read_ahead)
  jeśli nie wczytano to
    alokuj pamięć na stronę
  koniec jeśli
  sprawdź czy strona jest w page cache
  jeśli strona jest obecna w page cache to
    zwolnij zaalokowaną pamięć
    jeśli no_share to
      alokuj pamięć na kopię
      kopiuj stronę
    koniec jeśli
    sprowadź stronę do pamięci (w razie gdyby była wyswapowana)
    RETURN strona
  w przeciwnym przypadku
    wczytaj stronę z pliku (inode->i_op->readpage)
    dodaj wczytaną stronę do page cache
    jeśli no_share to
      alokuj pamięć na kopię
      kopiuj stronę
    koniec jeśli
    sprowadź stronę do pamięci (w razie gdyby była wyswapowana)
    RETURN strona
  koniec jeśli
koniec jeśli
```

- `sync`jako `filemap_sync`,
- `nopage` jako `filemap_nopage`,
- `swapout` jako `filemap_swapout`.

Funkcję `nopage` już opisaliśmy. Jest pewna różnica w jej działaniu: w przypadku próby sprowadzenia strony odpowiadającej obszarowi za końcem pliku dla obszarów dzielonych do procesu wysyłany jest sygnał `SIGBUS`, natomiast dla obszarów prywatnych zwraca stronę pustą *null page*(jak przy mapowaniu obszarów anonimowych).

Opis pozostałych funkcji:

`int` **filemap_swapout**`(struct vm_area_struct *vma, unsigned long offset, pte_t *page_table)`

Funkcja zapisuje stronę, która jest usuwana z pamięci fizycznej, do pliku, który ta strona mapuje. Parametry:

- `vma` – właściwa struktura `vm_area_struct`;
- `offset` – offset w pliku, do którego funkcja zapisze stronę;
- `pte_t` – wskaźnik na tablicę stron.

`int` **filemap_sync**`(struct vm_area_struct *vma, unsigned long address, size_t size, unsigned int flags)`

Parametry :

- `vma` – właściwa struktura VMA;
- `address` – adres początkowy obszaru do synchronizacji;
- `size` – rozmiar obszaru;
- `flags` – zmienna ta musi przyjmować jedną z flag:
  - `MS_SYNC` - odwołanie synchroniczne,
  - `MS_ASYNC` - odwołanie do strony jest asynchroniczne - dokonuje się poza procesorem,
  - `MS_INVALIDATE`;

Funkcja dokonuje synchronizacji zawartości obszaru pamięci z odpowiednimi fragmentami dysku. Wywołuje ona kaskadowo analogiczne funkcje na poziomie katalogu tablic stron i tablic stron aż wreszcie na poziomie pojedynczej strony, gdzie sprawdza czy strona była już przywołana do pamięci,wartość flag oraz to czy strona była zmieniana. Jeśli była, dokonuje zapisu strony na dysk.

`static void` **filemap_unmap**`(struct vm_area_struct *vma, unsigned long start, size_t len)`

Działanie tej funkcji sprowadza się do wywołania `filemap_sync(vma, start, len, MS_ASYNC)`, a więc zapisania do pliku całego mapowanego obszaru.

1.7 Zadania

1. Odpowiedz: O ile może się zmienić liczba obszarów VMA procesu po pomyślnym wywołaniu przez niego funkcji `do_mmap()`. Wskazówka: Czy odpowiedzi zależą od tego czy użyto flagi `MAP_FIXED`?

   a) może się nie zmienić b) może wzrosnąć o 1 c) może wzrosnąć o 2 d) może zmaleć o 1 e) może zmaleć o 2 f) może zmaleć o 17 g) może wzrosnąć o 22 h) zmaleć do 0

2. Odpowiedz o ile może się zmienić liczba obszarów VMA procesu po pomyślnym wywołaniu przez niego funkcji `do_munmap()`.

   a) może się nie zmienić b) może wzrosnąć o 1 c) może wzrosnąć o 2 d) może zmaleć o 1 e) może zmaleć o 2 f) może zmaleć o 17 g) może wzrosnąć o 22 h) zmaleć do 0

3. Zapalony programista Jaś dowiedział się właśnie o istnieniu funkcji `mmap()` i `munmap()`. Szybko włączył swój ulubiony edytor mcedit napisał “`#include <sys/mman.h>`” i już ma trzy programiki: prywatek.c, dzielonek.c i czytelniczek.c. Jaś szybko stworzył plik literka.txt i początkowo wypełnił go 3200 literami “J”. Program czytelniczek.c nie mapuje pliku literka.txt, lecz czyta go funkcją read i wypisuje pierwszy jego znak. Program prywatek.c mapuje plik literka.txt na sposób prywatny do pisania i czytania. Program dzielonek.c mapuje go też do pisania i czytania, ale na sposób dzielony. Prywatek i dzielonek, pomijając rodzaj mapowania, są niemal identyczne, różnią się tylko tym, że pierwszy na początku pliku literka.txt umieszcza “P” a drugi “D”.

   Kod programiku dzielonek.c to:

```
int main()
{
  int fd,i;
  char *bufor=0;
  if(!(fd=open("literka.txt",O_RDWR))){
    printf("Plik!!\n");
    exit(0);
  }
  if((bufor = (char*)mmap(bufor,25,PROT_READ | PROT_WRITE,MAP_SHARED,fd, 0)) == MAP_FAILED){
    printf("FAILED\n"); exit(0);
  }
  printf("Wskaznik: %p\n",bufor);
  printf( "Zawartosc: \n");
  putchar(bufor[0]);
  putchar('\n');
  printf("Teraz modyfikacja\n");
  bufor[1] = 'd';
  munmap(bufor,25);
  close(fd);
  return(0);
}
```

Odpowiedz:

(a) Czym na pewno różni się kod programu prywatek.c.

(b) Jaką wartość pierwszej literki pliku literka.txt odczyta czytelniczek jeśli zostanie uruchomiony. a) po programie prywatek b) po programie dzielonek

(c) Co mogło by się mogło stać gdyby Jaś użył flagi `MAP_FIXED`. Czy Ty użyłbyś jej w tym programie?

(d) Jak na działanie programu dzielonek wpłynęłaby zmiana ostatniego argumentu funkcji `mmap` (określa offset w pliku).

4. Dla każdego z podanych poniżej zestawów obszarów VMA odpowiedz czy możliwe jest by w strukturze procesu były przechowywane jednocześnie poniższe obszary VMA. Dla ustalenia uwagi przy części pól zapisano "jest OK", czyli że wartość w tym polu nie przeszkadza w jednoczesnym współistnieniu obszarów w procesie. Załóżmy że stała `PAGE_SIZE` nie wynosi 4096, lecz 1000.

   Zestaw 1

```
  struct vm_area_struct {
         struct mm_struct * vm_mm;    jest OK
         unsigned long vm_start;      1000
         unsigned long vm_end;        2000
         pgprot_t vm_page_prot;       jest OK
         unsigned short vm_flags;     jest OK
         short vm_avl_height;         0
         struct vm_area_struct * vm_avl_left;   null
         struct vm_area_struct * vm_avl_right;  null
         struct vm_area_struct *vm_next_share;  null
         struct vm_area_struct **vm_pprev_share; null
         struct vm_operations_struct * vm_ops;  wskaznik_do_operacji_typu1
         unsigned long vm_offset;                100
         struct file * vm_file;                  wskaznik_do_struktur_pliku_1
         unsigned long vm_pte;
};

   struct vm_area_struct {
         struct mm_struct * vm_mm;    jest OK
         unsigned long vm_start;      2000
         unsigned long vm_end;        3000
         pgprot_t vm_page_prot;       jest OK
         unsigned short vm_flags;     jest OK
         short vm_avl_height;         0
         struct vm_area_struct * vm_avl_left;   null
         struct vm_area_struct * vm_avl_right;  null
         struct vm_area_struct *vm_next_share;  null
         struct vm_area_struct **vm_pprev_share; null
```

```
        struct vm_operations_struct * vm_ops;  wskaznik_do_operacji_typu1
        unsigned long vm_offset;                100
        struct file * vm_file;                  wskaznik_do_struktur_pliku_1
        unsigned long vm_pte;           jest OK
 };
```

Zestaw 2

```
struct vm_area_struct {
        struct mm_struct * vm_mm;    jest OK
        unsigned long vm_start;      1000
        unsigned long vm_end;        3000
        pgprot_t vm_page_prot;       jest OK
        unsigned short vm_flags;     jest OK
        short vm_avl_height;         0
        struct vm_area_struct * vm_avl_left;   null
        struct vm_area_struct * vm_avl_right;  null
        struct vm_area_struct *vm_next_share;  jest OK
        struct vm_area_struct **vm_pprev_share; jest OK
        struct vm_operations_struct * vm_ops;  wskaznik_do_operacji_typu1
        unsigned long vm_offset;                0
        struct file * vm_file;                  wskaznik_do_struktur_pliku_1
        unsigned long vm_pte;
 };

   struct vm_area_struct {
        struct mm_struct * vm_mm;    jest OK
        unsigned long vm_start;      2000
        unsigned long vm_end;        3000
        pgprot_t vm_page_prot;       jest OK
        unsigned short vm_flags;     jest OK
        short vm_avl_height;         0
        struct vm_area_struct * vm_avl_left;   null
        struct vm_area_struct * vm_avl_right;  null
        struct vm_area_struct *vm_next_share;  jest OK
        struct vm_area_struct **vm_pprev_share; jest OK
        struct vm_operations_struct * vm_ops;  wskaznik_do_operacji_typu1
        unsigned long vm_offset;                1000
        struct file * vm_file;                  wskaznik_do_struktur_pliku_1
        unsigned long vm_pte;           jest OK
 };
```

Zestaw 3

```
  struct vm_area_struct {
```

```
        struct mm_struct * vm_mm;    jest OK
        unsigned long vm_start;      1500
        unsigned long vm_end;        3000
        pgprot_t vm_page_prot;       jest OK
        unsigned short vm_flags;     jest OK
        short vm_avl_height;         0
        struct vm_area_struct * vm_avl_left;   null
        struct vm_area_struct * vm_avl_right;  null
        struct vm_area_struct *vm_next_share;  jest OK
        struct vm_area_struct **vm_pprev_share; jest OK
        struct vm_operations_struct * vm_ops;  wskaznik_do_operacji_typu1
        unsigned long vm_offset;                0
        struct file * vm_file;                  wskaznik_do_struktur_pliku_1
        unsigned long vm_pte;
};

  struct vm_area_struct {
        struct mm_struct * vm_mm;    jest OK
        unsigned long vm_start;      3000
        unsigned long vm_end;        8000
        pgprot_t vm_page_prot;       jest OK
        unsigned short vm_flags;     jest OK
        short vm_avl_height;         0
        struct vm_area_struct * vm_avl_left;   null
        struct vm_area_struct * vm_avl_right;  null
        struct vm_area_struct *vm_next_share;  jest OK
        struct vm_area_struct **vm_pprev_share; jest OK
        struct vm_operations_struct * vm_ops;  wskaznik_do_operacji_typu1
        unsigned long vm_offset;                1000
        struct file * vm_file;                  wskaznik_do_struktur_pliku_1
        unsigned long vm_pte;           jest OK
};
```

Zestaw 4

```
 struct vm_area_struct {
        struct mm_struct * vm_mm;    jest OK
        unsigned long vm_start;      1500
        unsigned long vm_end;        3000
        pgprot_t vm_page_prot;       jest OK
        unsigned short vm_flags;     jest OK
        short vm_avl_height;         0
        struct vm_area_struct * vm_avl_left;   null
        struct vm_area_struct * vm_avl_right;  null
        struct vm_area_struct *vm_next_share;  jest OK
        struct vm_area_struct **vm_pprev_share; jest OK
```

```
        struct vm_operations_struct * vm_ops;   wskaznik_do_operacji_typu3
        unsigned long vm_offset;                0
        struct file * vm_file;                  null
        unsigned long vm_pte;
};

  struct vm_area_struct {
        struct mm_struct * vm_mm;     jest OK
        unsigned long vm_start;       3000
        unsigned long vm_end;         8000
        pgprot_t vm_page_prot;        jest OK
        unsigned short vm_flags;      jest OK
        short vm_avl_height;          20
        struct vm_area_struct * vm_avl_left;    wskaznik_do_lewego
        struct vm_area_struct * vm_avl_right;   wskaznik_do_prawego
        struct vm_area_struct *vm_next_share;   jest OK
        struct vm_area_struct **vm_pprev_share; jest OK
        struct vm_operations_struct * vm_ops;   wskaznik_do_operacji_typu1
        unsigned long vm_offset;                1000
        struct file * vm_file;                  wskaznik_do_struktur_pliku_1
        unsigned long vm_pte;            jest OK
};
```

5. Dlaczego struktura AVL drzewa obszarów VMA jest incjowana dopiero gdy liczba tych obszarów sięgnie `AVL_MIN_MAP_COUNT` (stałej równej 32 w naszej wersji jądra)?

1.8 Rozwiązania zadań

1. z flagą `MAP_FIXED` / bez flagi a) tak / tak b) tak / tak c) nie / tak d) tak / tak e) tak / nie f) tak / nie g) nie / nie h) nie / nie Uzasadnienie: a) Przykładowo gdy nastąpi połączenie z sąsiednim wcześniej zmapowanym obszarem. b) Chyba jasne. c) Gdy mapujemy z flagą `MAP_FIXED` obszar wewnątrz już wcześniej zmapowanego obszaru,wówczas powstają dwa obszary ze starego i jeden nowy w środku. d) Gdy zamapujemy dziurę między dwoma obszarami i możemy je połączyć w jeden. e) Z flagą `MAP_FIXED` gdy mamy trzy A<B<C obszary i zamapujemy obszar między cały odcinek między końcem A i początkiem C (odmapowany zostanie wówczas obszar B), może się zdarzyć (jakie są warunki?), że nowy obszar i obszary A i C dadzą się połączyć w jeden. f) Z flagą `MAP_FIXED` gdy mapujemy może się zdarzyć, że odmapujemy całkowicie 18 starych obszarów. g)W obu przypadkach może wzrosnąć co najwyżej o 1. h)Proces bez obszaru VMA?

2. a) tak b) tak c) nie d) tak e) tak f) tak g) nie h) raczej nie Uzasadnienie: a) Przykładowo gdy odmapowano tylko początek jednego obszaru VMA. b) Gdy odmapowano środek pewnego obszaru VMA. c) Można odmapować (w jednym wywołaniu) funkcji `do_munmap()` co najwyżej jeden środek obszaru VMA. A tylko wtedy ilość obszarów wzrasta. d) Chyba jasne. e) i f) Odmapowujemy odpowiednią ilość leżących jeden za drugim obszarów VMA. h) Jakiś obszar VMA powinien zawierać kod...

3. (a) Proces prywatek ma mapować plik w sposób prywatny, czyli musi użyć zamiast flagi `MAP_SHARED` flagę `MAP_PRIVATE`.

   (b) a) Proces prywatek zmienia pierwszy znak tylko dla siebie. W momencie zapisu strona przechowująca modyfikowany znak jest kopiowana. Czytelniczek, ani żaden inny niż prywatek proces, nie widzi więc zmian dokonanych przez proces prywatek.Czytelniczek wypisze "J". b) Modyfikacje dokonane przez proces dzielonek będą natychmiast widoczne dla wszystkich pozostałych procesów.Czytelniczek wypisze więc "D".Zmaiana dokonana przez dzielonka będzie naniesiona na dyskowy zapis pliku.

   (c) Jaś pokazał tu się od dobrej strony, flaga `MAP_FIXED` zmusza do przydziału dokładnie tego adresu wirtualnego. Są jednak sytuacje kiedy nie można przydzielić tego adresu. Może być zajęty na ważne dla programu informacje, funkcja `mmap` zwróci wtedy błąd. Trudno przykładowo odmapować kod programu, czy jego stos. Tu Jaś nie miałby żadnego pożytku z przydzielenia jakiegoś konkretnego adresu.

   (d) Offset w pliku określa punkt startowy, czyli miejsce od którego mapujemy plik.Jeśli byłby równy n, to dzielonek mapowałby plik od (n+1) pozycji, czyli wypisywałby, a potem zmieniał nie pierwszy a (n+1) znak.

4. Zestaw 1

   Odpowiedź: Nie, nie jest to możliwe. To się zgadza: Ta sama część pliku może być mapowany pod róznymi adresami. Obszary się nie przecinają. Nie da się połączyć w jeden obszar gdyż ofsett nie będzie się zgadzał. To się zgadza:nie jest zaincjalizowana lista obszarów związanych z tym samym inodem pliku (`vm_next_share` i `vm_pprev_share`)

   Zestaw 2

   Odpowiedź: Nie, nie jest to możliwe. Obszary te powinny być połączone,mapują ten sam plik,sąsiadują ze sobą i offsety się zgadzają.

   Zestaw 3

   Odpowiedź: Nie, nie jest to możliwe. To się nie zgadza: Obszary VMA nie mogą się nakładać na siebie, nawet jeśli tak jak w tym przypadku mapowanie "zgadza się" na przecięciu. Obszary VMA są zawsze rozłączne. Do tego pierwszy obszar zaczyna się w środku strony. Koniec i początek obszaru musi być wielokrotnością strony – obszary są przecież przechowywane w stronach pamięci fizycznej.

   Zestaw 4

   Odpowiedź: Nie, nie jest to możliwe. Jest dobrze: W pierwszej strukturze występuje mapowanie anonimowe, dlatego wskazanie do pliku jest null-em. Jest źle: Spójność AVL,jest ona wspólna dla wszystkich obszarów pamięci VMA procesu, a tu w jedym obszarze jest zaincjalizowana – a w drugim nie

5. Obszary VMA procesu są zawsze połączone posortowaną listą, która umożliwia wyszukiwanie, wstawianie i usuwanie obszaru VMA w czasie liniowym (małym, gdy lista jest krótka). Drzewa AVL pozwalają na wykonanie tych operacji w czasie logarytmicznym. Są to jednak zależności asymptotyczne. Dla małych struktur nie opłaca się incjalizować struktury AVL drzewa. Różnica między log(n) a n jest zbyt mała.

Literatura

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

[M.Beck i inni 99] : *LINUX KERNEL – jądro systemu.* Warszawa, MIKOM 1999.

[Studenci UW 97/98] : *Projekt LabLinux (1997/1998).* http://rainbow.mimuw.edu.pl/SO/LabLinux.

[Studenci UW 99/00] : *Projekt Podręcznik Linuxa (1999/2000).* http://rainbow.mimuw.edu.pl/SO/LinuxPodrecznik.