Spis treści

Rozdział 3

Zarządzanie pamięcią

3.1 Wstęp

Autorzy:

- Anna Majkowska
- Tomasz Fujak
- Marek Futrega
- Piotr Kozbial
- Piotr Sankowski

W rozdziale znajduje się opis zarządzania pamięcią w systemie Linux (jądro w wersji 2.2.12). Rozdział zaczyna się przedstawieniem wsparcia sprzętowego na procesorze Intel 386, a przede wszystkim mechanizmu podziału pamięci na ramki. Metoda ta opisana jest dokładniej w kolejnym podrozdziale, który koncentruje się na wykorzystywanym do zarządzania ramkami algorytmie buddy. Trzeci podrozdział przedstawia obsługę plików wykonywalnych ze szczególnym uwzględnieniem mechanizmów zarządzania pamięcią w czasie `exec` i `fork`, oraz mechanizmu **kopiowania przy zapisie** (ang. *copy on write*. Następne podrozdziały 5 i 6 prezentują mechanizmy pamięci dzielonej oraz mapowania plików do pamięci. W podrozdziałach 7,8,9 i 10 przechodzimy do wyjaśnienia struktury pamięci wirtualnej procesu oraz sposobu jej realizacji, przedstawiając używane struktury danych oraz algorytm `brk`. Podrozdział 11 koncentruje się na wymianie stron, omawia czynności wykonywane przy wymiatanu stron na dysk, oraz przy odwołaniach do stron nie obecnych w pamięci. Opisuje mechanizm wyboru strony do wymiecienia oraz demona `kswapd`. Ostatnia część tego podrozdziału porusza temat pamięci buforowej stron. W podrozdziale 12 zawarte są informacje na temat obsługi urządzeń wymiany. W punkcie 13 rozdziału przedstawiamy mechanizm przydziału pamięci dla jądra. Podrozdział 14 to kilka zadań poruszających tematykę rozdziału.

3.2 Wsparcie sprzętowe na procesorach Intel 386

3.2.1 Mechanizm tłumaczenia adresów

W systemie Linux adres wirtualny ulega tłumaczeniu na adres fizyczny w dwóch krokach. Najpierw tłumaczony jest na adres liniowy poprzez segmentację, a następnie poprzez stronicowanie na adres fizyczny.

Segmentacja

Adresy segmentowe pozwalają na otrzymywania adresu liniowego z adresu logicznego (wirtualnego). Każdy segment w systemie opisany jest przez ośmiobajtowy deskryptor segmentu, który zawiera niezbędne informacje (adres liniowy, wielkość, typ, prawa).

Segmenty dzielimy na dwa rodzaje:

1. **Regularne** (*regular segments*), na które składają się segmenty kodu i danych.
2. **Systemowe** (*system segments*), w których wyróżniamy:
    - *Task State Segments* (TSS), czyli **segmenty stanu procesu**
    - *Local Descriptor Tables* (LDT), czyli **lokalne tablice deskryptorów**

Rejestry segmentowe (`cs`, `ds`, etc.) zawierają **selektory** właściwych segmentów. Format selektora przedstawia poniższa tabela:

Tablica 3.1: Format selektora segmentu

bity:	15..3	2	1..0
opis:	indeks	TI	RPL

gdzie:

- TI - *Table Indicator* (**wskaźnik tablicy**):
  0 - indeks odwołuje się do globalnej tablicy deskryptorów
  1 - indeks odwołuje się do lokalnej tablicy deskryptorów
- RPL - **poziom praw**; Linux wykorzystuje tylko dwa poziomy:
  0 - poziom jądra
  1 - poziom użytkownika

Lokalne tablice deskryptorów nie są jednak standardowo wykorzystywane ponieważ zwykle zawierały takie same wartości dla wszystkich procesów: 1 - segment kodu, 2 - segment danych.

Deskryptory segmentów opisują każdy segment w pamięci. Format deskryptora przedstawia poniższa tabela:
gdzie:

Tablica 3.2: Format deskryptora segmentu

bity:	63-54	55	54	53	52	51-48	47	46	45	44-40	39-16	15-0
opis:	adres liniowy bity 31-24	G	D	R	U	wielkość bity 19-16	P	DPL	S	Typ	adres liniowy bity 23-0	wielkość bity 15-0

- P - obecność segmentu w pamięci fizycznej (1-tak, 0-nie)
- DPL - poziomy praw: 0-poziom jądra, 1-poziom użytkownika
- S - rodzaj segmentu: 1-systemowy, 0-normalny (danych/kodu)
- G - 1 oznacza granulację 4K (w Linuksie zawsze równy 1)
- R - zarezerwowany
- U - definiowalne przez programistę
- Typ - zależne od systemu

Stronnicowanie

Pamięc wirtualna podzielona jest na równe kawałki ustalonej wielkości nazywane **stronami**. (wielkość zdefiniowa jest w pliku `page.h` i przeważnie jest potęgą dwójki – zazwyczaj 4K). Pamięć fizyczna podzielona jest na kawałki takiej samej wielkośći nazywane **ramkami**.

Tłumaczeniem adresów zajmuje się *Memory Management Unit* (MMU) czyli **sprzętowa jednostka zarządzania pamięcią**.

Na procesorach Intel 386 dostępne są katalogi stron, które wskazują na tablice stron (chociaż Linux wykorzystuje trzy poziomy, to na i386 tablice środkowe wskazują na jedną i tą samą tablicę wewnetrzną).

Elementy tablicy wewnętrznej zawierają oprócz adresu ramki w pamięci fizycznej także dodatkowe informacje jak np. czy ramka jest w pamięci, jak jest chroniona, czy zostało coś do niej zapisane, etc. (dokładny opis w dalszej części).

Każdy proces ma własny katalog tablic stron. Kiedy proces jest wznawiany, jądro ładuje do rejestru CR3 adres fizyczny (!) katalogu tablic stron tego procesu. Format adresu liniowego przedstawia poniższa tabela:

Tablica 3.3: Format adresu liniowego

bity:	31..22	21..12	11..0
opis:	katalog	tablica	offset

Fizyczny adres wyznacza się następująco:

`CR3` + `KATALOG` daje `adres_tablicy_stron`
`adres_tablicy_stron` + `TABLICA` daje `adres_ramki`
`adres_ramki` + OFFSET daje `adres fizyczny`

Adresy katalogów stron (tablic stron) są wyrównane do wielkości strony a 12 najmłodszych bitów jest wykorzystywane do przechowywania informacji o tablicy stron (stronie) wskazywanej przez element tablicy.

Format elementu katalogu/tablicy stron przedstawia poniższa tabela:

Tablica 3.4: Format elementu katalogu/tablicy stron

bity:	31..12	11..9	8	7	6	5	4	3	2	1	0
opis:	ADRES	OS	0	0	D	A	0	0	U/S	R/W	P

gdzie:

- D - dirty page (strona brudna, czyli coś zostało do niej zapisane od ostatniego wczytania z dysku)
- R/W - możliwość zapisu przez użytkownika (1-tak, 0-nie)
- U/S - 1 oznacza stronę użytkownika, 0 stronę jądra
- P - czy obecna w pamięci fizycznej (1-tak, 0-nie i reszta bitów wskazuje położenie na dysku)
- A - 1 oznacza, że strona była czytana/zapisana (*accessed*); zerowane przy starzeniu strony
- OS - do wykorzystania przez system (np. do LRU, etc.)

Stronicowanie jest włączane przez ustawienie najwyższego bitu rejestru `CR0`. Przy każdym dostępie do strony sprawdzane są prawa dostępu i przy ich naruszeniu lub nieobecności strony w pamięci wystepują **błędy strony** (*page faults*). Sterowanie jest wtedy przekazywane do jądra, które wczytuje stronę z dysku albo robi co trzeba. W momencie zdarzenia, rejestr `CR2` zawiera liniowy adres instrukcji, która spowodowała ostatni błąd strony. Znaczenie bitów w kodzie błędu strony przedstawia poniższa tabela:

Tablica 3.5: Format kodu błędu strony

bit	wartość 0	wartość 1
0	strona niedostępna	błąd poziomu ochrony
1	błąd przy odczycie	błąd przy zapisie
2	tryb jądra	tryb użytkownika

3.2.2 Pamięci podręczne

W procesorach Intel 386 istnieje *Translation Lookaside Buffer* (TLB), czyli **bufor translacji bliskiego otoczenia**. Zawiera on adresy *fizyczne* ramek ostatnio używanych adresów logicznych (liniowych) stron. Kiedy trzeba uzyskać adres fizyczny jakiejś strony, procesor najpierw patrzy do TLB czy nie ma tam tej informacji. Jesli nie ma, to wykonywanych jest kilka odwolań do pamięci tzn. do katalogu oraz tablicy stron i dopiero odczytywany jest adres fizyczny strony. Bez TLB pojedyncze odwołanie do pamięci wymagałoby trzech wcześniejszych odwołań co byłoby **bardzo** czasochłonne.

TLB jest czyszczone przy zmianie wartości adresu `CR3` oraz przy przełączaniu zadań (co powoduje zmianę rejestru `CR0`). W Linuksie jest także czyszczone bezpośrednio przez wywołanie procedury `flush_tlb()`, która tylko ładuje ponownie wartość rejestru `CR3`.

3.3 Struktura pamięci fizycznej

3.3.1 Ramki i tablice ramek

Pamięć fizyczna w systemie Linux jest podzielona na ramki, które pokrywają całą pamięć fizyczną. Wielkość ramek jest specyficzna dla poszczególnych atchitektur. Tabela

Tablica 3.6: Wielkość ramek dla różnych architektur

Alpha	ARM	Intel 386	M68k	MIPS	PPC	SPARC	SPARC64
8192	4096[a]/32k[b]	4096	4096	4096	4096	4096/8192[c]	8192

[a]procesory arm6/arm7/sa110
[b]procesry arm2/arm250/arm3
[c]SUN4

Definicja wielkości ramki pamięci znajduje się w pliku `include/asm-<platforma>/page.h`

```
#define PAGE_SIZE (1UL << PAGE_SHIFT)}
```

Czyli właściwie to stała PAGE_SHIFT określa wielkości ramki jako

$PAGE\_SIZE := 2^{PAGE\_SHIFT}$

Oprócz tych dwóch stałych mamy stałą definiującą maskę strony

```
#define PAGE_MASK (~(PAGE_SIZE-1))}
```

Spójrzmy jeszcze na definicję makra przeliczającego adres w kontekscie jądra na numer ramki

```
#define MAP_NR(addr) (((unsigned long)(addr) - PAGE_OFFSET) >> PAGE_SHIFT)
```

Stała `PAGE_OFFSET` określa rozmiar pamięci użytkownika, pamięć o większym adresie jest przeznaczona dla jądra. Na pierwszy rzut oka takie przeliczenie wydaje się dość dziwne. Początek fizycznego adresu 0xA0000 (jest to adres pamięci karty graficznej dla architektury Intel 386) leży w ramce o numerze 160, a adres logiczny wskazujący tą ramkę to `PAGE_OFFSET + 0xA0000` czyli dzięki takiemu przeliczeniu zapewniliśmy, że pamięc karty graficznej znajduje się w opszarze pamięci jądra. Dzięki takiemu przeliczeniu również obszar portów znalazł się poza obszarem użytkownika.

Informacje o ramkach pamięci znajdują się w tablicy `mem_map` przechowującej rekordy typu `struct page` zdefiniowane w `include/linux/mm.h`.

```
typedef struct page {
        struct page *next;
        struct page *prev;
        struct inode *inode;
        unsigned long offset;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;
        struct wait_queue *wait;
        struct page **pprev_hash;
        struct buffer_head * buffers;
} mem_map_t;
```

Znaczenia poszczególnych pól:

- `next` i `prev` - tworzą listę cykliczną wolnych obszarów o takim samym rozmiarze gdy ramka jest wolna lub listę cykliczną ramek do których zamapowany jest plik.
- `inode` i `offset` - wskazują na i-węzeł reprezentujący zamapowany plik do pamięci oraz przesunięcie w tym pliku.
- `next_hash` i `prev_hash` - jest to lista haszująca służąca do szybszego wyszukiwania ramek należących do danego i-węzła.
- `count` - pole to mówi ile procesów używa danej ramki albo, jeśli ramka należy do i-węzła, to liczba przyłączeń danego i-węzła plus jeden jeśli ramka ma bufury.
- `wait_queue` - kolejka procesów czekających aż operacja wejścia/wyjścia wykonywana na tej ramce zostanie zakończona.
- `buffer_head` - cykliczna lista buforów zaalokowanych to tej ramki.
- `flags` - flagi opisujące stan ramki
  - `PG_locked` - ustawiony bit oznacza, że na ramce wykonywana jest operacja WE/WY. Bit ten zapobiega sytuacji, w której mogłyby być wykonywane jednocześnie dwie operacje WE/WY na ramce.
  - `PG_error` - flaga informuje czy w czasie operacji WE/WY na ramce wystąpił błąd.
  - `PG_referenced` - flaga oznacza, że do ramki było odwołanie, jest ustawiana w funkcji `try_to_swap_out` próbującej wymieść zadaną stronę gdy jest ustawiony bit *accessed* w tablicy stron.
  - `PG_dirty` - flaga nie używana a tej wersji systemu, system używa bezpośrednio analogicznej flagi z tablicy stron.
  - `PG_uptodate` - czy ramka zawiera aktualne dane. Gdy ramka jest wczytana z dysku bit jest ustawiany o ile nie wystąpił błąd.
  - `PG_free_after` - ramka będzie bezzwłocznie zwolniona po zakończeniu operacji zapisu ramki.
  - `PG_decr_after` - oznacza, że na ramce wykonywana jest asynchroniczna operacja wymiany razem z ustawieniem zwiększany jest licznik asynchronicznych operacji wymiany. Po zakończeniu operacji WE/WY, bit jest zerowany oraz zmniejszany jest licznik asynchronicznych operacji.
  - `PG_swap_unlock_after` - bit ustawiany gdy wykonywana jest operacja We/Wy na cache pamięci wymiany (bit ustawiany gdy jest ustawiony bit `PG_swap_cache`)
  - `PG_DMA` - na ramce może być wykonywana operacja DMA. Bit ten jest ustawiany w podczas startu systemu dla wszystkich ramek w funkcji `free_area_init` plik mm/page_all.c dopiero potem w części zależnej od architektury w funkcji `mem_init` plik `arch/<platforma>/mm/init.c` jest odpowiednio czyszczony.
  - `PG_Slab` - ramka jest przeznaczona na pamięć cache dla jądra
  - `PG_swap_cache` - oznacza, że ramka należy do cache pamięci wymiany

– `PG_skip` - używane dla architektur sparc/sparc64 w celu pominięcia określonych obszarów adresowych
– `PG_reserved` - flaga oznacza, że do ramki nie mogą następować żadne odwołania. Flaga jest ustawiana tak samo jak flaga `PG_DMA` przy starcie sytemu dla wszystkich ramek, a następnie zerowana w części zależnej od architektury.

3.3.2 Wolne ramki

Informacja o wolnych ramkach w systemie przechowywana jest w tablicy `free_area` zdefiniowanej w pliku `mm/page_alloc.c`. Definicja ta wygląda następująco:

```
struct free_area_struct {
       struct page *next;
       struct page *prev;
       unsigned int * map;
};

static struct free_area_struct free_area[NR_MEM_LISTS];
```

Stała `NR_MEM_LISTS` dla wszystkich komputerów wynosi 10 oprócz AP1000, który na niektóre bufory potrzebuje 8 megabajtów pamięci ciągłej.

W elemencie `free_area[i]` przechowywana jest informacja o wolnych blokach rozmiaru 2^i ramek pamięci. Wolne bloki o określonej wielkości tworzą listę cykliczną powiązaną polami `next` i `prev` elementu `freearea[i]` oraz polami `next` i `prev` struktury `page`. Listę pustą tworzy element `freearea[i]` powiązany sam ze sobą, czyli zainicjowany przez funkcję z `mm/page_alloc.c`

```
static inline void init_mem_queue(struct free_area_struct * head)
{
        head->next = memory_head(head);
        head->prev = memory_head(head);
}
```

Gdzie `memory_head(head)` to po prostu wskaźnik na `head`:

```
#define memory_head(x) ((struct page *)(x))
```

Usuwanie i wstawianie do tej listy jest zrealizowane następująco:

```
static inline void
add_mem_queue(struct free_area_struct * head, struct page * entry)
{
        struct page * next = head->next;

        entry->prev = memory_head(head);
        entry->next = next;
        next->prev = entry;
        head->next = entry;
}
```

```
static inline void remove_mem_queue(struct page * entry)
{
        struct page * next = entry->next;
        struct page * prev = entry->prev;
        next->prev = prev;
        prev->next = next;
}
```

Wskaźnik map struktury free_area_struct wskazuje na mapę bitową w której dla każdej pary bloków bliźniaczych przeznaczony jest jeden bit. Bit ten jest ustawiony tylko wtedy gdy jeden z tych bloków jest wolny a drugi częsciowo lub całkowicie zajęty.
Zmienić wartość bitu dla ramki o numerze map_nr i wielkości 2^i ramek można ciągiem instrukcji:

```
        struct free_area_struct *area = free_area + order;
        unsigned long index = map_nr >> (1 + order);
        unsigned long mask = (~0UL) << order;

        map_nr &= mask;
        nr_free_pages -= mask;
        change_bit(index, area->map))
```

Proces init

Po starcie komputera żadna z tych tablica nie istnieje nie ma więc mowy o korzystaniu z nich przy alokacji pamięci, sposób powstawania tych tablic przebiega następująco w procedurze start_kernel plik init/main.c

- `setup_arch(&command_line, &memory_start, &memory_end);`
  funkcja z arch/<platforma>/kernel/setup.c inicjującej zależne od architektury dane systemu. Zwrócone wartości memory_start i memory_end wskazują początek i koniec dostępnej pamięci.

- `memory_start = paging_init(memory_start,memory_end);`
  funkcja z arch/<platforma>/mm/init.c tworzy katalogi i tablice stron wartość a następnie wywołuje funkcję free_area_init z mm/page_alloc.c inicjującą tablice mem_map oraz free_area. Wartość memory_start jest zwiększana najpierw o pamięć potrzebną na przechowanie tablic i katalogów stron a następnie o rozmiar mapy bitowej.

- `memory_start = console_init(memory_start,memory_end);`
  inicjacja konsoli

- `memory_start = kmem_cache_init(memory_start, memory_end);`
  inicjacja pamięci podręcznej jądra (plik mm/slab.c.

- `mem_init(memory_start,memory_end);`
  inicjacja zależna od architektury arch/<platforma>/mm/init.c, inicjowane są obszary DMA i obszary zarezerwowane.

Pamięć nie zainicjowana

Pamięć przydzielana procesom jest zawsze wyzerowana. Nie następuje to wcale poprzez zapisanie do tego obszaru pamięci zer. W czasie inicjacji systemu proces `init` alokuje jedną ramkę pamięci, która następnie jest zerowana. Adres tej ramki można uzyskać makrem `ZERO_PAGE` plik `include/asm-<platforma>/pgtable.h` makro to mimo że pobiera jeden argument nie korzysta z niego. Strony obszarów nowo przydzielanej pamięci są wszystkie mapowane na tą jedną ramkę, przy tym mapowaniu stosowana jest technika COW (ang. *copy on write*). Spójrzmy na kod odpowiedzialny za to kopiowanie (plik `mm/memory.c`:

```
static inline void copy_cow_page(unsigned long from, unsigned long to)
{
        if (from == ZERO_PAGE(to)) {
                clear_page(to);
                return;
        }
        copy_page(to, from);
}
```

Efektem zastosowania takiego rozwiązania jest:

- pamięć fizycznie przydzielona jest rzeczywiście wykorzystywana
- odczyt z pamięci nie zainicjowanej nie powoduje jej przydzielenia
- pamięć nie zainicjowana jest wyzerowana

3.3.3 Zwalnianie i zajmowanie ramek

System Linux wykorzystuje algorytm buddy (bliźniaków) przy przydzielaniu spójnych bloków pamięci. W wyniku ich wielokrotnego przydzielania i zwalniania dochodzi do fragmentacji, czyli powstawania wolnych bloków niedużych rozmiarów rozrzuconych po całej pamięci.

Algorytm buddy pozwala na uniknięcie fragmentacji, dzięki łączeniu ze sobą odpowiednich, zwolnionych bloków pamięci. W celu przyśpieszenia wyszukiwania tych bloków stosuje się specjalne mapy bitowe (pole `map` struktury `free_area` opisanej w p.3.3.2.

Zajmowanie bloków pamięci

Przydzielane są spójne bloki wielkości 2^i ramek, gdzie i=0,.., `NR_MEM_LISTS`-1. Numerem danego bloku jest numer pierwszej ramki w tym bloku. Zajmowanie bloków pamięci obsługuje funkcja `__get_free_pages` (`mm/page_alloc.h`):

```
unsigned long __get_free_pages(int gfp_mask, unsigned long order)
```

Funkcja ta stara się, jeśli to możliwe, przydzielić blok pamięci wielkości 2^{order}. Gdy znajdzie taki, wówczas zwraca jego adres, wpp zwraca 0.

Algorytm 1 __get_free_pages

Dane wejściowe: order (rozmiar bloku=2^{order})
Wynik: adres przydzielonego bloku

jeśli order >= `NR_MEM_LISTS` **to**
  zwróć zero, żądany blok był zbyt duży
koniec jeśli
znajdź najmniejszy wolny blok, niemniejszy od żądanego (przeszukując listy `free_area[i].mem_map_t` dla i=order,..,`NR_MEM_LISTS`-1) - niech ma on rozmiar 2^k
jeśli nie ma takiego bloku **to**
  to wywołaj funkcje `try_to_free_pages`
koniec jeśli
B:=znaleziony blok
usuń blok **B** z listy `free_area[k].mem_map_t`
zmień bit w `free_area[k].map[nr_bloku div 2]` na przeciwny
dopóki znaleziony blok jest większy od żądanego (k>order) **wykonuj**
  podziel blok na dwie połowy (dwóch bliźniaków)
  pierwszą połowę wstaw na listę `free_area[k-1].mem_map_t`
  zmien na przeciwny bit w tablicy `free_area[a-1].map[nr_bloku div 2]`
  B:=druga połowa bloku **B**
koniec dopóki
zwróć adres uzyskanego bloku

Jeśli żądano bloku pamięci z DMA (ustawiony bit `__GFP_DMA` w `gfp_mask`), wówczas funkcja `__get_free_pages` przeszukuje tylko bloki pamieci obsługującej DMA, w przeciwnym przypadku funkcja najpierw szuka bloku bez DMA, a jeśli takiego nie znajdzie, wówczas szuka dowolnego bloku.

Zwalnianie bloków pamięci

Algorytm zwalniania bloków pamięci jest odwrotny do algorytmu ich zajmowania - następuje łączenie rodzielonych bliźniaków.

Do zwalniania bloków pamięci i ich odpowiedniego łaczenia używana jest funkcja `free_pages`:

`void free_pages(unsigned long addr, unsigned long order)`

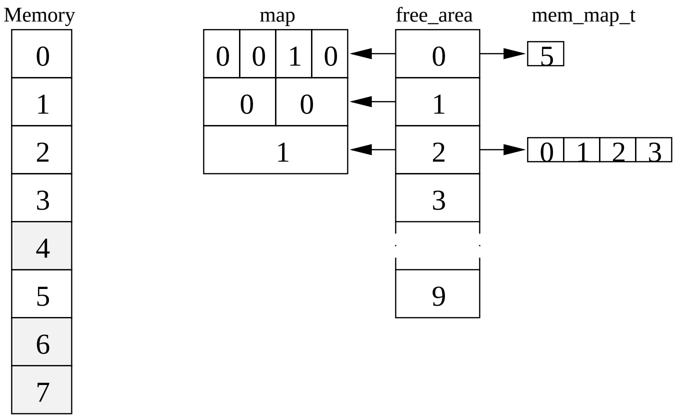
Jej argumentami są adres bloku oraz jego rozmiar (tzn. rozmiar=2^{order}).

Algorytm 2 free_pages

Dane wejściowe: addr (adres bloku), order (rozmiar bloku=2^{order})

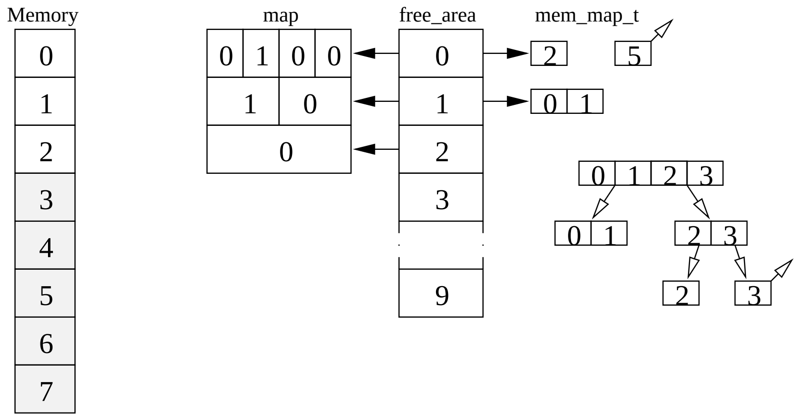
```
jeśli blok jest współdzielony (parametr count>1) to
   zakończ działanie
koniec jeśli
zmniejsz o 1 liczbe użytkowników
koniec:=false
B:=zwalniany blok
dopóki not koniec wykonuj
   zmień bit w free_area[order].map[nr_bloku B div 2] na przeciwny
   jeśli bit ten był przed zmianą równy 0 to
      wstaw blok B na listę free_area[order].mem_map_t
      koniec:=true
   w przeciwnym przypadku
      znajdź bliźniaka bloku B
      usuń go z listy i połącz z blokiem B
      B:=połączony blok
      zwiększ order o 1
   koniec jeśli
koniec dopóki
```

Przykład



Rysunek przedstawia wygląd tablicy `free_area`, list bloków i map bitowych dla przykładowego stanu pamięci. Komórki pamięci w kolorze szarym są zajęte, natomiast białe są wolne. Ponieważ listy bloków dla elementów tablicy `free_area` od 3 do 9 są puste, więc część tej tablicy została obcięta. Na listach wolnych bloków znajduje się jeden blok wielkości jednej ramki i jeden blok wielkości czterech ramek.

Na poniższym rysunku przedstawiony jest stan po przydzieleniu dwu bloków jednoramkowych.



Gdy przychodzi pierwsze żądanie bloku rozmiaru jeden, algorytm znajduje taki wolny blok (o numerze 5) i go przydziela. Zmienia bit w `free_area[0].map[2]` na 0.

Gdy przychodzi drugie żądanie bloku tej samej wielkości, najmniejszym blokiem, niemniejszym od żądanego, jest blok rozmiaru cztery (blok 0-3 o numerze 0). Zostaje on usunięty z listy, zaktualizowany zostaje bit w `free_area[2].map[0]`.

Usunięty blok jest dzielony na dwie połowy (0-1 i 2-3). Pierwsza połowa (0-1) zostaje wstawiona do listy `free_area[1].mem_map_t`, bit w `free_area[1].map[0]` zmienia się na przeciwny.

Blok 2-3 jest powtórnie dzielony. Powstały z podziału blok 2 jest wstawiany na listę `free_area [0].mem_map_t` i zmienia się bit w `free_area [0].map[1]`. Drugi blok powstały z podziału (o numerze 3) zostaje przydzielony.

Gdy blok o numerze 3 zostanie zwolniony, wykona się proces odwrotny. Ponieważ w `free_area [0].map[1]`=1 zostanie odnaleziony bliźniak bloku (blok o numerze 2) i zostaną one połączone w blok 2-3 (o numerze 2). Bit `free_area[0].map[1]` zostanie ustawiony na 0.

Dla bloku 2-3 algorytm także odnajdzie bliźniaka - będze to 0-1. Bloki zostaną połączone w blok 0-3, zaktualizowany będzie bit w `free_ area[1].map[0]`.

Ponieważ blok 0-3 nie ma już bliźniaka na listach - `free_area[2].map [0]`=0, to blok zostaje wstawiony na listę `free_area[2].mem_map_t`. Bit w `free_area[2].map[0]` zostaje ustawiony na 1.

3.4 Pliki wykonywalne

3.4.1 Obsługa różnych formatów plików wykonywalnych

System Linux umożliwia uruchamianie wielu różnych formatów plików wykonywalnych, zarówno binarnych, jak i skryptowych. Obok formatów standardowych, obsługiwanych domyślnie, istnieje możliwość definiowania nowych, niestandardowych formatów i rejestrowania ich w systemie. Pliki ładowalne do pamięci można ogólnie podzielić na dwie kategorie: programy (których załadowanie wiąże się z utworzeniem nowego procesu), oraz biblioteki dzielone, które zawierają kod przeznaczony do dynamicznego łączenia.

Formaty standardowe

Standardowo obsługiwane są następujące formaty plików wykonywalnych:

a.out — prosty format plików binarnych, wypierany obecnie przez format ELF,

ELF — nowszy format plików binarnych, bardziej skomplikowany i dający większe możliwości,

java — obecnie istnieje (wspierana przez jądro systemu) możliwość uruchamiania programów w Javie,

script — wszelkiego rodzaju skrypty, obsługiwane na równych prawach z plikami binarnymi.

Opis formatu binarnego

Do opisu formatu binarnego (określenie to obejmuje tu również formaty skryptowe) służy struktura linux_binfmt, zdefiniowana w pliku nagłówkowym include/binfmts.h. Zawiera ona pola będące wskaźnikami do funkcji obsługujących podstawowe operacje na danym formacie binarnym:

*int (*load_binary) (...)* funkcja ładująca program wykonywalny,

*int (*load_shlib) (int fd)* funkcja ładująca bibliotekę dzieloną,

*int (*core_dump) (...)* funkcja zapisująca na dysku obraz pamięci (*core*), używana w wypadku wystąpienia błędu.

Podczas ładowania pliku binarnego (w funkcji *exec*) wywoływane są odpowiednie funkcje (*load_binary* lub *load_shlib*) dla kolejnych znanych (zarejestrowanych) formatów plików binarnych, tak długo aż format pliku zostanie rozpoznany przez którąś z nich, lub zostaną wyczerpane wszystkie znane formaty (w tym wypadku plik uznaje się za niepoprawny i ładowanie kończy się niepowodzeniem).

Standardowo obsługiwane formaty plików są zdefiniowane w następujących strukturach *linux_binfmt*:

format a.out struktura aout_format (w pliku *binfmt_aout.c*),

format ELF struktura elf_format (w pliku *binfmt_elf.c*),

format java struktura java_format (w pliku *binfmt_java.c*),

format script struktura script_format (w pliku *binfmt_script.c*).

Rejestracja formatów niestandardowych

Aby system zaczął rozpoznawać i obsługiwać niestandardowy format plików binarnych, należy:

1. Zdefiniować dla niego strukturę typu *linux_binfmt* wraz z odpowiednimi funkcjami.
2. Zarejestrować nowy format binarny przy pomocy funkcji *register_binfmt*:
   int register_binfmt(struct linux_binfmt)*
   której parametrem jest zdefiniowana przez nas struktura opisująca nowy format pliku binarnego.

Jeśli później ten format przestanie być potrzebny, możemy go wyrejestrować przy pomocy funkcji *unregister_binfmt*:
int unregister_binfmt(struct linux_binfmt)*
której parametrem jest ta sama struktura, której użyliśmy do zarejestrowania formatu.

3.4.2 Format a.out

Format a.out jest prostym formatem plików binarnych, wypieranym obecnie przez nowszy i bogatszy format ELF. Format a.out obejmuje zarówno programy jak i biblioteki dzielone. Kod zawarty w pliku jest relokowany podczas ładowania. Zwykle oznacza to modyfikowanie wszystkich odwołań i skoków do bezwzględnych adresów w pamięci zgodnie z rzeczywistym rozmieszczeniem segmentów programu w pamięci wirtualnej. Nie da się tego uniknąć w systemie Linux, w którym standardowo nie stosuje się translacji segmentów procesu (tzn. adresy logiczne wewnątrz segmentów nie zaczynają się od zera). Istnieje także możliwość relokacji danych. Dla celów uruchamiania (debugging) i analizy kodu format a.out zapewnia możliwość umieszczenia w pliku binarnym tablicy symboli.

Plik binarny w formacie a.out składa się z następujących części (w kolejności występowania w pliku):

1. Nagłówek.
2. Segment kodu wykonywalnego (text).
3. Segment danych (data).
4. Tablica relokacji dla segmentu kodu.
5. Tablica relokacji dla segmentu danych.
6. Tablica symboli.

Nagłówek

Nagłówek pliku binarnego w formacie a.out jest zdefiniowany w pliku include/asm/a.out.h jako struktura struct exec. Zawiera ona następujące pola:

a_info znaczniki, ważniejsze opisano poniżej,

a_text długość segmentu kodu (text),

a_data długość segmentu danych (data), *a_bss* wielkość obszaru zmiennych niezainicjowanych (bss),

a_syms wielkość tablicy symboli,

a_entry adres punktu wejścia do programu,

a_trsize wielkość tablicy relokacji dla segmentu kodu,

a_drsize wielkość tablicy relokacji dla segmentu danych.

Ważniejsze znaczniki zawarte w polu *a_info* to:

MACHTYPE docelowa platforma sprzętowa, na której kod może się wykonać,

MAGIC typ pliku a.out:

- *OMAGIC* plik obiektowy,
- *NMAGIC* "czysty" plik wykonywalny,
- *ZMAGIC* plik wykonywalny ze stronicowaniem na żądanie,
- *QMAGIC* plik wykonywalny ze stronicowaniem na żądanie, pierwsza strona pamięci nie jest mapowana, co umożliwia wykrywanie odwołań do niezainicjowanych wskaźników (gdyż mają one wartość 0).

Tablica relokacji dla segmentu kodu i danych

Tablica relokacji zarówno dla segmentu kodu, jak i dla segmentu danych ma ten sam format. Jest to tablica struktur typu *relocation_info*, zdefiniowanych w pliku *include/linux/a.out.h*. Struktura ta zawiera informacje o jednym elemencie (polu) podlegającym relokacji. Oto ważniejsze pola tej struktury:

r_address adres relokowanego pola,

r_length długość pola w postaci logarytmu dwójkowego (tzn. np. wartość 3 oznacza pole 8-bajtowe),

r_symbolnum znaczenie tego pola zależy od zawartości pola *r_extern*,

r_extern typ relokacji:

- 0: relokacja o adres początku segmentu podanego symbolicznie w *r_symbolnum* (przy pomocy stałych *N_TEXT*, *N_DATA*, itd.),
- 1: relokacja o wartość symbolu o indeksie *r_symbolnum* w tablicy symboli.

Tablica symboli

Tablica symboli jest tablicą struktur typu *nlist*, zdefiniowanych w pliku *include/linux/a.out.h*. Pojedyncza struktura zawiera informację o jednym symbolu. Oto jej najważniejsze pola:

n_type typ symbolu,

n_value wartość symbolu.

3.4.3 Format ELF

Wstęp

Wśród formatów plików wykonywalnych dostepnych dla systemów uniksowych głównymi standardami sa COFF (common object file formay) i ELF (executable and linking format) - format podstawowy dla systemu Linux. Jest (ELF) formatem dość ogólnym, pozwalajacym na swobode w ustawianiu platformy sprzętowej, typu pliku i zawartości pamięci w momencie uruchamiania pliku; wystarczajaco jednak szczegółowym, by pliki rzczywiście sie uruchamiały, i nie bylo watpliwosci, co mozna i należy w pliku umieścić, a co nie jest dopuszczone przez standard.

Symbole relokowalne wymagają przemieszczenia (dodania wartości przesunięcia), relatywne - podania przesunięcia w/g początku segmentu, a dynamiczne - złączenia z pliku zewnętrznego (po nazwie) - wpisania adresu bezwzglednego w pamięci wirtualnej.

W nagłówkach są zawarte: informacje pozwalajace zidentyfikować plik jako instancje formatu ELF, określenie typu pliku, informacje o architekturze docelowej, definicje segmentów, dane dla relokacji i linkowania oraz tablice symboli.

Zawartość pliku

1. Nagłówek ELF
2. Tablica nagłówków segmentów - dla execa, dla biblioteki opcjonalna
3. Segmenty (lub sekcje)
4. Informacje dodatkowe
5. Tablica naglówków sekcji - dla biblioteki, dla execa opcjonalna

Technikalia

Oto nagłówek pliku elf (odmiana 32 bit.):
`typedef struct elf32_hdr{`

`unsigned char e_ident[EI_NIDENT];` - napis magiczny do rozpoznawania, jego postac to: 0x7f, "ELF", 32/64 bity, LSB/MSB, wersja i wypełnienie do końca tablicy.

`Elf32_Half e_type;` - typ pliku, ze względu na rodzaj kodu:

`ET_NONE` - pusty
`ET_REL` - relokowalny
`ET_EXEC` - czysto wykonywalny
`ET_DYN` - ładowany dynamicznie
`ET_CORE` - zrzut pamięci do pliku
`ET_LOPROC, ET_HIPROC` - procedury.

`Elf32_Half e_machine;` - platforma sprzętowa:

zadna, MIPS 32, 3000, 4000, SPARC - kilka typów, i386, i486, MC68k, MC88k, i860,Alpha.

```
  Elf32_Word   e_version;           - wersja pliku
  Elf32_Addr   e_entry;             - start programu
  Elf32_Off    e_phoff;             - offset nagłówka programu
  Elf32_Off    e_shoff;             - offset wektora sekcji
  Elf32_Word   e_flags;             - flagi pliku
  Elf32_Half   e_ehsize;            - rozmiar nagłówka ELFa
  Elf32_Half   e_phentsize;         - rozmiar elementu w tablicy phdr
  Elf32_Half   e_phnum;             - ilosc elementów w tablicy phdr
  Elf32_Half   e_shentsize;         - rozmiar elementu w tablicy sekcji
  Elf32_Half   e_shnum;             - ilosc elementów w tablicy sekcji
  Elf32_Half   e_shstrndx;          - index w sekcji hdr dla stringów
} Elf32_Ehdr;
```

Najwazniejsze z punktu widzenia uruchamiania sa segmenty zawarte w programie, oto struktury definiujące segment:

```
typedef struct elf32_phdr{
    Elf32_Word   p_type;     typ segmentu
    Elf32_Off    p_offset;   offset w pliku
    Elf32_Addr   p_vaddr;    adres wirualny w pamięci procesu
    Elf32_Addr   p_paddr;    adres fizyczny (??)
    Elf32_Word   p_filesz;   rozmiar w pliku
    Elf32_Word   p_memsz;    rozmiar w pamięci
    Elf32_Word   p_flags;    flagi (prawa)
    Elf32_Word   p_align;    wyrównanie
  } Elf32_Phdr;
```

Typ segmentu: `PT_NULL` - pusty (koniec listy)
`PT_LOAD` - ładowany przy starcie
`PT_DYNAMIC` - ładowany dynamicznie
`PT_INTERP` - systemowa biblioteka dzielona
`PT_NOTE` - opis sekcji
`PT_SHLIB` - biblioteka dzielona
`PT_PHDR` -nagłówek programu
`PT_LOPROC, PT_HIPROC` - procedury

Typ pliku - architektura:

zadna, MIPS 32, 3000, 4000, SPARC - kilka typów, i386, i486, MC68k, MC88k, i860,Alpha.

Definicja sekcji:

```
typedef struct elf32_shdr {
    Elf32_Word sh_name;      Nazwa sekcji - indeks do tablicy stringów
    Elf32_Word sh_type;      Typ sekcji - jeden z nastepujących:
```

```
   SHT_NULL, SHT_PROGBITS,
   SHT_SYMTAB - tablica symboli
   SHT_STRTAB - tablica stringów
   SHT_RELA - tablica symboli relatywnych
   SHT_HASH,
   SHT_DYNAMIC - tablica symboli dynamicznych
   SHT_NOTE - notka (informacja tekstowa)
   SHT_NOBITS
   SHT_REL - tablica relokacji
   SHT_SHLIB - biblioteki dzielone
   SHT_DYNSYM - tablica symboli dynamicznych
   SHT_NUM, SHT_LOPROC, SHT_HIPROC, SHT_LOUSER, SHT_HIUSER;
     Elf32_Word sh_flags;        Flagi sekcji (prawa)
     Elf32_Addr sh_addr;         Wirtualny adres sekcji przy wykonaniu
     Elf32_Off sh_offset;        Offset sekcji w pliku
     Elf32_Word sh_size;         Rozmiar sekcji
     Elf32_Word sh_link;         Wskaźnik do nastepnej sekcji
     Elf32_Word sh_info;
     Elf32_Word sh_addralign;    Wyrównanie
     Elf32_Word sh_entsize;      Jeśli w sekcji jest tabela, to rozmiar jej elementu
} Elf32_Shdr;
```

obiekt relokowalny:

```
typedef struct elf32_rel {
     Elf32_Addr    r_offset;  Pozycja, do ktorej przyłożyć funktor
     Elf32_Word    r_info;    Indeks i typ relokacji
} Elf32_Rel;
```

relatywny:

```
typedef struct elf32_rela{
     Elf32_Addr    r_offset;    Pozycja, do ktorej przyłożyć funktor
     Elf32_Word    r_info;      Indeks i typ relokacji
     Elf32_Sword   r_addend; Stała dodawana do symbolu
} Elf32_Rela;
```

i symbol w tablicy symboli:

```
typedef struct elf32_sym{
     Elf32_Word st_name;      Nazwa symbolu, indeks do tablicy nazw
     Elf32_Addr st_value;      Wartość
     Elf32_Word st_size;       Rozmiar
     unsigned char st_info;        Typ, atrybuty łączenia;
```

sposob rozdzielenia danych:

```
#define ELF32_ST_BIND(x) ((x) >> 4)
```

```
#define ELF32_ST_TYPE(x) (((unsigned int) x) & 0xf)
```
typy moga byc:
`STT_NOTYPE, STT_OBJECT, STT_FUNC, STT_SECTION, STT_FILE;`
`unsigned char st_other;` Bez znaczenia, 0
`Elf32_Half st_shndx;` Indeks do właść. sekcji
`} Elf32_Sym;`

Typy relokacji (używane w `elf??_rel[a]` - funktory) - `R_arch_NAME`, np:
`R_386_NONE` - brak akcji,
`R_386_32` - bezpośrednia relokacja 32-bitowa,
`R_386_PC32` - bezpośrednia relokacja 32-bitowa w segemncie kodu,
`R_386_RELATIVE` offset
`R_386_GOTOFF` adres (offset w segmencie)
`R_386_GOTPC` adres (w kodzie)
`R_386_NUM` numer obiektu

Notka w sekcji `PT_NOTE`:
```
typedef struct elf32_note {
    Elf32_Word    n_namesz;   rozmiar nazwy (długość)
    Elf32_Word    n_descsz;   rozmiar zawartości
    Elf32_Word    n_type;     typ zawartosci
} Elf32_Nhdr;
```

3.4.4 Fork

Podczas tworzenia nowego procesu za pomocą funkcji fork() wykonywane są następujące czynności:

1. Utworzenie nowej struktury *task_ struct* dla procesu.
2. Utworzenie nowego stanu kontekstu jądra.
3. Przydzielenie nowego identyfikatora procesu.
4. Kopiowanie zasobów procesu macierzystego.
5. Utworzenie nowej struktury *mm_ struct*.
6. Utworzenie nowego katalogu stron, lub podłączenie katalogu stron procesu macierzystego.
7. "Kopiowanie" pamięci wirtualnej.

Spośród tych czynności cztery ostatnie dotyczą bezpośrednio zagadnienia zarządzania pamięcią i zostaną omówione w kolejnych punktach tego rozdziału.

Kopiowanie zasobów

Kopiowanie zasobów procesu macierzystego do procesu potomnego jest wykonywane przez funkcję *do_fork*. Funkcja *fork*, wywołując funkcję *do_fork*, przekazuje jej parametr *clone_flags* typu *unsigned long*, składający się z jednobitowych znaczników (flag), decydujących o tym, które zasoby mają zostać skopiowane. Dla tych znaczników przewidziano symboliczne stałe o nazwach postaci *CLONE_ **, gdzie gwiazdka jest nazwą zasobu. Do kopiowania większości zasobów funkcja *do_fork* posługuje się pomocniczymi funkcjami o nazwach postaci *clone_ **, gdzie, jak poprzednio, gwiazdka jest nazwą zasobu. Poniżej opisano poszczególne typy zasobów podlegających kopiowaniu, oraz odpowiednie flagi.

Kopiowanie otwartych plików

Kopiowanie otwartych plików jest kontrolowane przez flagę *CLONE_FILES* i ewentualnie wykonywane przez funkcję *copy_files*. Jeśli flaga *CLONE_FILES* jest ustawiona, to proces potomny otrzyma nowe kopie deskryptorów otwartych plików (początkowo identyczne z deskryptorami procesu macierzystego), w przeciwnym razie procesy macierzysty i potomny będą dzieliły wspólne deskryptory otwartych plików.

Kopiowanie systemów plików

Kopiowanie podłączonych systemów plików jest kontrolowane przez flagę *CLONE_FS* i wykonywane przez funkcję *copy_fs*.

Kopiowanie procedur obsługi sygnałów

Jeżeli flaga *CLONE_SIGHAND* jest ustawiona, to proces potomny otrzyma kopię procedur obsługi sygnałów procesu macierzystego, w przeciwnym razie proces potomny będzie miał początkowo ustawioną standardową obsługę wszystkich sygnałów. Kopiowaniem obsługi sygnałów zajmuje się funkcja *copy_sighand*.

Powielenie identyfikatora procesu

Jeżeli flaga *CLONE_PID* jest ustawiona, to proces potomny będzie miał ten sam identyfikator (PID), co proces macierzysty, w przeciwnym razie proces potomny otrzyma nowy identyfikator. W obu przypadkach te proste czynności są wykonywane bezpośrednio w funkcji *do_fork*, bez użycia funkcji pomocniczej.

Kopiowanie pamięci wirtualnej

Flaga *CLONE_VM* decyduje o tym, czy nowy proces będzie samodzielnym (ciężkim) procesem, czy też wątkiem (lekkim procesem). Jeżeli flaga *CLONE_VM* jest ustawiona, to nowy proces otrzymuje własną kopię pamięci wirtualnej, tzn. nową strukturę *mm_struct* i tablicę stron. Otrzymujemy tzw. ciężki proces, z własną przestrzenią danych. Same dane nie są jednak od razu kopiowane z pamięci procesu macierzystego do pamięci procesu potomnego - stosuje się tu tzw. technikę *Copy On Write*, opisaną bardziej szczegółowo dalej w tym rozdziale. Jeżeli flaga *CLONE_FM* jest wyzerowana, to nowy proces dzieli pamięć wirtualną z procesem macierzystym i wszelkie modyfikacje pamięci będą widoczne w obu procesach. Taki proces potomny nazywamy wątkiem (lekkim procesem).

Proces kopiowania pamięci wirtualnej można podzielić na trzy etapy, opisane w kolejnych punktach: kopiowanie lub współdzielenie struktury *mm_struct*, kopiowanie lub współdzielenie katalogu stron oraz logiczne kopiowanie przestrzeni danych.

Poniższa tabela zawiera podsumowanie flag kopiowania i kontrolowanych przez nie zasobów.

flaga	funkcja wykonująca kopiowanie	opis
CLONE_FILES	copy_files()	kopiowanie otwartych plików
CLONE_FS	copy_fs()	kopiowanie systemów plików
CLONE_SIGHAND	copy_sighand()	kopiowanie obsługi sygnałów
CLONE_PID	-	potomek otrzymuje PID rodzica
CLONE_VM	copy_mm()	kopiowanie pamięci wirtualnej

Tworzenie nowej struktury mm_struct

Jeśli nowy proces jest wątkiem (flaga *CLONE_VM* jest wyzerowana), to struktura *mm_struct* jest współdzielona z procesem macierzystym - po prostu przepisuje się wskaźnik do niej i zwiększa licznik odwołań.

Jeśli flaga *CLONE_VM* jest ustawiona, to dla procesu potomnego tworzy się nową kopię struktury *mm_struct*.

Kopiowanie katalogu stron

Wątki (flaga *CLONE_VM* wyzerowana) dzielą katalog stron z procesem macierzystym, a zatem wystarczy ustawić dla procesu potomnego wskaźnik na tablicę stron procesu macierzystego.

Samodzielne procesy (flaga *CLONE_VM* ustawiona) otrzymują własną kopię katalogu stron. Pola tego katalogu są początkowo wypełnione zerami. Ich alokacja następuje przy pierwszym odwołaniu. Obszar pomiędzy 3GB a 4GB jest odwzorowywany na pamięć jądra.

Kopiowanie pamięci wirtualnej

Ten etap jest wykonywany tylko dla samodzielnych procesów, tzn. wtedy, gdy flaga *COPY_VM* jest ustawiona. Zajmuje się tym funkcja *dup_mmap*. Nie powoduje ona rzeczywistego kopiowania danych, ewentualne kopiowanie, jeśli okaże się potrzebne, nastąpi później (patrz dalej - technika *Copy On Write*). Tutaj następuje tylko tworzenie odpowiednich struktur *vm_area_struct* (drzewa AVL opisujące spójne obszary pamięci wirtualnej) oraz "logiczne kopiowanie", zależne od typu obszarów pamięci:

1. Obszary pochodzące z i-węzła, są po prostu dołączane do kolejki FIFO związanej z danym plikiem.

2. Zwykłe obszary:

   (a) obszary pamięci dzielonej lub tylko do odczytu, pozostają współdzielone, gdyż nie ma możliwości konfliktu, następuje tylko zwiększenie licznika odwołań do obszaru,

(b) obszar zwykły (nie dzielony) z prawem zapisu, powinno się go skopiować, aby jeden proces nie zamazywał danych drugiego procesu, ale niekoniecznie trzeba to robić od razu - stosuje się tu technikę Copy On Write.

3.4.5 Technika Copy On Write

Jest to technika stosowana zamiast natychmiastowego kopiowania obszarów pamięci dla procesu potomnego. Jeśli proces potomny może zapisywać do jakiegoś obszaru pamięci, to oczywiście musi otrzymać własną jego kopię. Samo prawo do zapisu nie oznacza jednak jeszcze, że proces kiedykolwiek rzeczywiście takiego zapisu dokona. Kopiowanie zatem od razu wszystkich obszarów "zagrożonych" zapisem byłoby wysoce nieefektywne. Zamiast tego można pozwolić obu procesom na korzystanie ze wspólnej pamięci, dopóki tylko oba (lub wszystkie, gdyż ta sytuacja może dotyczyć więcej niż dwu procesów) procesy dokonują jedynie odczytów z tego obszaru. Kopiowanie zaś nastąpi dopiero, gdy jeden z procesów spróbuje dokonać zapisu. Na tym właśnie polega technika *Copy On Write*. A oto jak przebiega typowy scenariusz kopiowania obszaru pamięci z prawem do zapisu z procesu macierzystego do potomnego:

1. Podczas tworzenia procesu potomnego dla każdej strony rozpatrywanego obszaru zeruje się bit RW (bit prawa do zapisu), oraz ustawia znacznik COW (Copy On Write) i zwiększa licznik odwołań do strony.
2. W pewnym momencie jeden z procesów podejmuje próbę zapisu do pewnej strony tego obszaru. Ponieważ strona ma wyzerowany bit RW, procesor zgłasza wyjątek page fault.
3. Procedura obsługi wyjątku rozpoznaje flagę COW i traktuje tę sytuację nie jako błąd ochrony, ale właśnie jako przypadek opóźnionego kopiowania w technice COW.
4. Procedura obsługi sprawdza licznik odwołań do strony. Zachodzą dwie możliwości:
   (a) licznik odwołań jest większy od 1, wówczas następuje zmniejszenie licznika odwołań, skopiowanie strony, ustawienie bitu RW dla skopiowanej strony i udostępnienie kopii procesowi, który wywołał instrukcję zapisu powodującą wyjątek,
   (b) licznik odwołań jest równy 1, zatem jest to ostatni proces, który może zażądać zapisu do tej strony, można więc udostępnić mu tę stronę już bez kopiowania, ustawiany jest jedynie bit RW i usuwany znacznik COW.
5. Procedura obsługi wyjątku kończy się i procesor rozpoczyna wykonywanie instrukcji, która spowodowała wyjątek od początku.

3.4.6 Exec - zarządzanie pamięcią

Wstęp

Tak jak nowe procesy w systemach uniksowych powstają po wywołaniu funkcji `fork()`, to jedyną metodą uruchomienia nowego kodu jest wywołanie funkcji systemowej z rodziny `exec()`. Każda z nich jest interfejsem do funkcji `execve()`. Aby umożliwić uruchamianie plików w formatach nie uwzględnionych przez twórców systemu, a także zmodularyzować kod jądra, projektanci oparli proces uruchomieniowy na rejestrowanych w jądrze procedurach usługowych, które odpowiadają za zarówno rozpoznanie pliku po jego nagłówku, jak i - w przypadku rozpoznania - wczytanie zawartości i uruchomienie.

Realizacja

Uslugi uruchomieniowych

Operacje na usługach uruchomieniowych realizują funkcje:
`register_bin_fmt(struct linux_binfmt *)` i `unregister_bin_fmt()`, gdzie

```
struct linux_binfmt {
  struct linux_binfmt * next; - wskaźnik do reszty listy
  struct module *module;          - wskaźnik do modułu z opisywanymi funkcjami
  int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
- ładowanie pliku wykonywalnego
  int (*load_shlib)(int fd);    - ładowanie biblioteki dzielonej
  int (*core_dump)(long signr, struct pt_regs * regs);
- zrzut stanu procesu do pliku
```

Funkcja rejestrująca dołącza do listy w jądrze nową strukturę, charakteryzującą nowy format wykonywalny. W przypadku próby ponownej rejestracji zwracany jest błąd (`errno=EBUSY`). W ten sam sposób obsługiwane są standardowe formaty wykonywalne Linuksa: ELF, a.out, skrypty i java - właściwe dla nich funkcje są rejestrowane przy starcie systemu.

Funkcja `exec()` i pomocnicze

Funkcja systemowa `execve()` wywołuje funkcję `do_execve()`, która ustawia strony środowiska i argumentów w pamięci wirtualnej procesu, w funkcji `read_exec()` następuje sprawdzenie sensownoci próby odczytania pliku wykonywalnego - nie będzie on wczytany jeśli nie jest plikiem z flagą x dla właciciela, jest otwarty do zapisu, użytkownik nie ma praw do uruchomienia (a także w przypadku biblioteki dzielonej lub wywołania z flagą ptrace - śledzenia programu - do czytania). W przypadku stwierdzenia sensownoci wywołania, wczytywany jest nagłówek pliku (128 bajtów) i wywoływane po kolei funkcje z listy formatów, aż który z nich zwróci wartość różną od błędu `ENOEXEC` - co oznaczać będzie, że format został rozpoznany - albo wyczerpią się handlery. Chociaż dalsze postępowanie zależy od funkcji ładującej, i może być różne dla różnych formatów, to jednak pewne działania są wspólne, np.: usuwanie struktur poprzedniego programu lub ustawianie pamięci dla argumentów.

`int prepare_binprm(struct linux_binprm *bprm)` - sprawdza możliwość uruchomienia pliku (względy bezpieczeństwa i sensowność operacji na pliku).

`int sys_uselib(const char * library)` - wczytuje z podanego pliku bibliotekę dzieloną i "uruchamia" ją.

Do tej operacji jest używany standardowy mechanizm ładowania.

`unsigned long setup_arg_pages(unsigned long p, struct linux_binprm * bprm)` - ustawia w strukturach procesu pamięć dla argumentów i zmiennych środowiskowych - przeznaczone są na to 32 górne (nad pamięcią stosu) strony. Używany jest przy tym mechanizm alokatora tafli.

`int read_exec(struct dentry *dentry, unsigned long offset, char * addr, unsigned long count, int to_kmem)` - ta funkcja realizuje wczytanie z pliku do pamięci.

`int exec_mmap(void)` - jeśli proces ma struktury na własność, to zwalnia struktury związane z pamięcią: pamięć cache, odblokowuje pamięć procesu, zwalnia segmenty i bufor stron bliskiego otoczenia. Inaczej przydziela nową strukturę zarządzania pamięcią, kopiuje ustawienia segmentów, tworzy katalog stron procesu i zwalnia (odblokowuje) pamięć procesu.

`int flush_old_exec(struct linux_binprm * bprm)` - usuwa wszelkie powiązania procesu z poprzednim programem: tworzy tablice obsługi sygnałów, usuwa starą mapę pamięci (exec_mmap()), zwalnia starą obsługę sygnałów, usuwa informację o wątkach i zamyka pliki z flagą close_on_exec.

`int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)` - dwukrotnie przechodzi listę zarejestrowanych formatów, próbując uruchomić podany plik. Rezultat inny niż -`ENOEXEC` oznacza rozpoznanie formatu (koniec przeszukiwania listy).

Skrypt jest rozpoznawany po tym, że pierwszymi znakami w pliku są #! , następna sekwencja (aż do białego znaku, jest nazwą interpretera, reszta linii to parametry. Tak więc po rozpoznaniu skryptu we wczytywanym pliku należy uruchomić plik o nazwie podanej po symbolach #!, z parametrami: nazwą skryptu (pierwsza linia usunięta) i resztą (po nazwie interpretera) pierwszej linii.

Aplikacje i aplety javy (początkowe symbole w plikach, to odpowiednio: *0xcafebabe* i "*<!-applet*") są traktowane podobnie jak skrypty - ale interpretera dla nich należy szukać pod pseudoplikami: `/usr/bin/java` i `/usr/bin/java-appletviewer`.

Nieco inaczej jest z plikami binarnymi - a.out i ELF. Jako binaria są bezporednio (bez programowej interpretacji) wykonywane przez procesor. Potrzebują jednak odpowiedniego środowiska - bibliotek C i schematu wywołań systemowych. Dlatego przy ładowaniu należy wykonać linkowanie - umiecić w pamięci wirtualnej procesu niezbędne dane. Format a.out obsługiwany jest z przyczyn historycznych, starsze programy są w tym włanie formacie; cechują go statyczne linkowanie i mała elastyczność zawartości. ELF jest obowiązującym dla Linuksa standardem; uwzględnia wielość odmian zależnych od architektury, środowiska i typu pliku.

Przykladowe wywolanie `exec()`

Oto schemat operacji podejmowanych podczas uruchamiania pliku przykładowego formatu binarnego (ELF).

`int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)` - funkcja systemowa execve(),

```
struct linux_binprm bprm;
struct dentry * dentry;

for (i=0 ; i<MAX_ARG_PAGES ;i++) bprm.page[i] = 0;     wyczyść tablicę stron

dentry = open_namei(filename, 0, 0);          odnajdź plik na dysku

retval = prepare_binprm(&bprm);         sprawdź, czy można uruchomić
```

```
  bprm.p=copy_strings(1,&bprm.filename,bprm.page,bprm.p,2);
  bprm.p = copy_strings(bprm.envc,envp,bprm.page,bprm.p,0);
bprm.p = copy_strings(bprm.argc,argv,bprm.page,bprm.p,0);
```
- ustaw argumenty i środowisko

```
  retval = search_binary_handler(&bprm,regs);
  if (retval >= 0) return retval;                sukces !
```

Inaczej coś poszło nie tak, zwolnij inode i strony przeznaczone dla argumentów.

Wewnątrz
`int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)` proces ładowania przebiega następująco:

```
  int try,retval=0;
  struct linux_binfmt *fmt;

  for (try=0; try<2; try++) {          - dwa (!) przejcia przez listę
  for (fmt = formats ; fmt ; fmt = fmt->next) {
  int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
  if (!fn) continue;                 - jeśli brak funkcji ładującej, to pomiń format
  retval = fn(bprm, regs);      - spróbuj wykonać
  if (retval >= 0) {                 - sukces
  current->did_exec = 1;
  return retval;}
  if (retval != -ENOEXEC) break;        - format rozpoznany; brak sukcesu
  if (!bprm->dentry) return retval;            - format nierozpoznany, dentry zniszczone
} //koniec petli for
  if (retval != -ENOEXEC) {                - format ok, nie powiodło się wykonanie
  break;
  }
  } return retval;
```

A teraz sedno - funkcja właściwa dla formatu ELF:
```
static inline int
do_load_elf_binary(struct linux_binprm * bprm, struct pt_regs * regs)
```
Sprawdzenie sensownoci pliku:
- pierwsze bajty 0x7f, "ELF";
- typ pliku - czysto wykonywalny (`ET_EXEC`) lub ładowany dynamicznie (`ET_DYN`);
- zgodność architektury pliku z zastanym sprzętem;
- system plików musi także umożliwiać mapowanie plików do pamięci;
- suma długości deklaracji segmentów niewiększa niż 64kB.

Wczytanie wspomnianych deklaracji (`read_exec()`).
Wyzerowanie danych o segmentach procesu
```
  elf_ppnt = elf_phdata;
```

```
elf_bss = 0;
...
end_data = 0;
```

Dla każdego segmentu w obrazie (pliku) powtarzaj:
jeśli segment jest bibliteką dzieloną:
jeśli wybrano już bibliotekę, to błąd
przydziel pamięć jądra na nazwę biblioteki,
odczytaj ją z pliku;
jeśli jest to `/usr/lib/libc.so.1` albo `/usr/lib/ld.so.1`, to jest to plik iBCS2, inaczej należy użyć natywnych bibliotek Linuksa.
otwórz plik biblioteczny (jeśli możesz) i wczytaj z niego 128 bajtów
sprawdź rodzaj pliku (biblioteka a.out, ELF czy błąd ?)
i ustaw go; w razie wątpliwoci wskaż format ELF.

Teraz trzeba ustawić argumenty(skopiować **argv do przydzielonych wczeniej stron i policzyć argc);
Usunąć resztki starego execa (flush_old_exec()),
Wyczycić pamięć wirtualną procesu

```
current->mm->end_data = 0;
current->mm->end_code = 0;
...
current->mm->mmap = NULL;
```

Ustawić miejsce startu

```
elf_entry = (unsigned long) elf_ex.e_entry;
```

i ustawić pochodzenie (wpływa na położenie stosu)

```
SET_PERSONALITY(elf_ex, ibcs2_interpreter);
```

wstaw strony z argumentami do przestrzeni adresowej

```
bprm->p = setup_arg_pages(bprm->p, bprm);
```

"pod" nimi ustaw stos

```
current->mm->start_stack = bprm->p;
```

a teraz sztuczka: trzeba przesunąć w elfach dynamicznie ładowanych początki segmentów ponad ewentualne biblioteki dzielone, które mogą być uruchomione później:

```
load_bias = ELF_PAGESTART(elf_ex.e_type==ET_DYN ? ELF_ET_DYN_BASE : 0);
```

dochodzimy do meritum: zmapujemy plik do pamięci:
Dla każdego segmentu w pliku powtarzaj:
jeśli nie jest typu ładowalnego (`elf_ppnt->p_type != PT_LOAD`), to go pomiń;
Ustaw flagi mapowania i adres w pamięci wirtualnej (odczytaj z pliku)
jeśli plik jest czyto wykonywalny (**pure executable**, `elf_ex.e_type == ET_EXEC`), to zafiksuj adresy przy mapowaniu.

Wszystko przygotowane, wykonaj mapowanie, przycinając i przesuwając odpowiednio (do granicy strony):

```
error = do_mmap(file, ELF_PAGESTART(load_bias + vaddr),
(elf_ppnt->p_filesz + ELF_PAGEOFFSET(elf_ppnt->p_vaddr)),
```

```
elf_prot, elf_flags, (elf_ppnt->p_offset - ELF_PAGEOFFSET(elf_ppnt->p_vaddr)));
```

Ustal, gdzie leży początek kodu, zaczyna i kończy się sterta i dane, gdzie jest brk.
Odłóż plik, to już koniec z nim.

Relokacje

Poprzestawiaj wszystkie cezury w pliku o przesunięcie (przskok przez ew. biblioteki dzielone i dosunięcie do granicy stron).

```
elf_entry += load_bias;
...
elf_brk += load_bias;
```

Teraz pora zająć się bibliotekami (jeśli są jakie). Sprowadza się to do wywołania `load_???_interp(...)`, gdzie ??? to aout albo elf. W przypadku porażki do procesu wysyłany jest sygnał `SIGSEGV`.

Odłóż jeszcze informacje o bibliotece dzielonej na stos:

```
create_elf_tables(...)
```

Ustaw informacje o poszczególnych obszarach:

```
current->mm->start_brk = current->mm->brk = elf_brk;
current->mm->end_code = end_code;
...
current->mm->start_stack = bprm->p;
```

Wywołanie set_brk() ustawia przy okazji strony bss i break (dane nieinicjowane i sterta).

```
set_brk(elf_bss, elf_brk);
```

Zainicjuj zerami dane nieinicjowane.

```
padzero(elf_bss);
```

Pliki o pochodzeniu (ang. **personality**) System V wersja 4 wymagają mapowania strony zerowej tylko do odczytu.

```
if ( current->personality == PER_SVR4 )
 error =
do_mmap(NULL, 0, 4096, PROT_READ | PROT_EXEC,MAP_FIXED | MAP_PRIVATE, 0);
```

Dalsza praca to ustawienie wątku (poprawienie informacji w tablicy procesów),

```
start_thread(regs, elf_entry, bprm->p);
```

i wyjście. Jeśli coś nie wyszło, to należy zwolnić pamięć struktur pomocniczych (ale nie programu uruchamianego - to zostanie zrobione przy wyjściu).

Uwagi

Jak jest zapewnione przebywanie w pamięci nie więcej niż jednej kopii programu na raz? Otóż przy mapowaniu pliku do pamięci w przypadku gdy już on tam jest, jest jedynie zwiększany licznik odwołań do struktury, bez ponownego przydzielania pamięci. Ten mechanizm, w połączeniu z dynamicznym ładowaniem (wczytywaniem do pamięci stron naprawdę potrzebnych - czyli tych, do których są odwołania) i mechanizmem *copy-on-write* (ale to przy fork) powoduje efektywne wykorzystanie pamięci dla kodu wykonywalnego (i danych - ale w mniejszym stopniu) w czasie uruchamiania nowych procesów i programów.

3.5 Mapowanie plików do pamięci

Autorzy:

- Piotr Miłkowski
- Ireneusz Roze
- Jarosław Tyszka
- Daniel Sadkowski
- Przemysław Siwik

Do odzwierciedlania binarnej postaci pliku dyskowego w przestrzeń adresową procesu używamy mechanizmu stronicowania plików, nazywanego też niekiedy mapowaniem. Najczęściej występującą sytuacją w której mechanizm stronicowania pliku znajduje zastosowanie, jest ładowanie przez jądro plików wykonywalnych funkcją `exec`, które zamiast wczytywać mapujemy do pamięci. W miarę potrzeb odpowiednie strony będą wczytywane w wyniku błędów braku strony. Aby sprawnie móc posługiwać się tym mechanizmem w strukturach pamięci wirtualnej muszą być umieszczone dane pozwalające szybko odnaleźć odpowiadający danej stronie pliku fragment. Do najbardziej istotnych struktur w stronicowaniu plików zaliczamy `task_struct` i `cache_page`. W `cache_page` zapamiętane są obecne w pamięci strony mapowanych plików. Ponadto w `vm_area_struct` przechowywane są niezbędne dane, bezpośrednio dotyczące poszczególnych obszarów pamięci, takie jak np. jego adres lub adres i-węzła, jeśli obszar jest częścią pliku. Programista może używać mapowania plików do własnych zastosowań za pomocą funkcji bibliotecznych:

- Mapowanie obszaru.

```
#ifndef __USE_FILE_OFFSET64
extern __ptr_t mmap __P ((__ptr_t __addr, size_t __len, int __prot,
int __flags, int __fd, __off_t __offset));
#else
extern __ptr_t mmap __P ((__ptr_t __addr, size_t __len, int __prot,
int __flags, int __fd, __off_t __offset))
__asm__ ("mmap64")
#endif

#ifdef __USE_LARGEFILE64
extern __ptr_t mmap64 __P ((__ptr_t __addr, size_t __len, int __prot,
int __flags, int __fd, __off64_t __offset));
#endif
```

Powyższe definicje oznaczają:

Mapowanie obszaru zaczynającego się ADDR o długości LEN bajtów z OFFSET z pliku o deskryptorze FD stosownie do PROT i FLAGS. Jeśli MAP_FIXED bit ustawiony we FLAGS, mapowanie będzie ściśle od adresu ADDR (musi być wyznacznikiem strony) w przeciwnym razie funkcja dokona konwersji na najbliższy poprawny adres. Zwracana wartością jest aktualny adres lub MAP_FAILED w przypadku błędu (ustawia errno).

Działanie funkcji:

Po wywołaniu sprawdza poprawność argumentów i wyszukuje odpowiedni obszar pamięci wirtualnej. Następuje allokacja vm_area_struct i wypełnienie jej podstawowymi wartościami oraz zwolnienie z pamięci obszarów o zachodzących adresach.

- Dealokacja mapowań.

  ```
  extern int munmap __P ((__ptr_t __addr, size_t __len));
  ```

  Dealokacja mapowań obszaru od ADDR długości LEN bajtów. Zwraca 0 w przypadku sukcesu, -1 gdy błąd (ustawia errno).

- Zmiana zabezpieczenia.

  ```
  extern int mprotect __P ((__ptr_t __addr, size_t __len, int __prot));
  ```

  Zmiana zabezpieczenia pamięci w obszarze zaczynającym się w ADDR i długości LEN bajtów na PROT. Zwraca 0 w przypadku sukcesu, -1 gdy błąd i ustawia errno.

- Synchronizacja obszaru.

  ```
  extern int msync __P ((__ptr_t __addr, size_t __len, int __flags));
  ```

- Zablokowanie stron mapowanych.

  ```
  extern int mlock __P ((__const __ptr_t __addr, size_t __len));
  ```

  Zablokowanie wszystkich stron mapowanych do obszaru [ADDR,ADDR+LEN].

- Odblokowanie stron mapowanych.

  ```
  extern int munlock __P ((__const __ptr_t __addr, size_t __len));
  ```

  Odblokowanie stron poprzednio mapowanych do obszaru [ADDR,ADDR+LEN].

- Zablokowanie stron procesu.

  ```
  extern int mlockall __P ((int __flags));
  ```

Spowodowanie, że wszystkie aktualnie mapowane strony procesu staną się rezydujące w pamięci aż do momentu, gdy się je odblokuje przy użyciu funkcji `munlockall`, proces się zakończy, lub proces wywoła funkcję `execve`.

- Odblokowanie stron procesu.

  ```
  extern int munlockall __P ((void));
  ```

  Wszystkie aktualne mapowania stron procesu w przestrzeni adresowej zostaną odblokowane.

Ponadto dostępne są funkcje systemowe: [1]

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

Funkcja `do_mmap` służy do mapowania plików do pamięci. Przydziela ona pamięć na nową strukturę `vm_area_struct`. Po wywołaniu funkcji następuje sprawdzenie parametrów i ewentualny powrót z błędem (ENODEV, EINVAL, ENOMEN, EAGAIN, EACCES).

Za pomocą funkcji `get_unmapped_area` wyszukuje odpowiedni obszar pamięci wirtualnej. Następuje alokacja `vm_area_struct` i wypełnienie jej pól podstawowymi wartościami. Przy użyciu funkcji `do_munmap` następuje zwolnienie obszaru po poprzednim mapowananiu. Uaktualnienie struktur realizuje funkcja `insert_vm_struct`. W celu połączenia bliskich obszarów następuje wywołanie funkcji `merge_segments`.

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

Funkcją o przeciwstawnym działaniu do poprzedniej jest funkcji `do_munmap`. Służy ona do odmapowywania plików z pamięci. Po sprawdzeniu poprawności argumentów wyszukuje wszystkie struktury `vm_area_struct` do których należy odmapowywany obszar i umieszcza je na specjalnej liście. Następuje zwolnienie pamięci używanej przez `vm_area_struct`, uaktualnienie pola `mm_strukt` i ewentualne zlepienie dziur powstałych w wyniku odmapowywania.

```
static int do_mlock(unsigned long start, size_t len, int on)
```

Funkcja `do_mlock` po sprawdzeniu poprawność argumentów, odpowiednio zmienia wartość pola flagi w strukturze `vm_area_struct` i dokonuje scalenia ewentualnie ujednoliconych obszarów.

3.6 Wirtualna przestrzeń adresowa procesu.

Przestrzeń adresowa każdego procesu podzielona jest na tzw. obszary pamięci wirtualnej (Virtual Memory Areas). Każdy taki obszar może mieć inaczej ustawione różne atrybuty ochrony (informacja dla jądra systemu operacyjnego, ponieważ dla jednostki zarządzającej pamięcią i tak wiążące są informacje zawarte w tablicy stron (Każdy obszar pamięci wirtualnej jednego procesu może mieć różne sposoby obsługi page_fault.)).

Strukturą danych opisującą pojedynczy obszar pamięci wirtualnej jest tt vm_area_struct zdefiniowanej w pliku `include/linux/mm.h`. Struktura ta zorganizowana jest w większe całości. Dla jednego procesu mamy więc:

[1] w `linux/mm/*`

- drzewo AVL uporządkowane względem początku obszaru (pola `vm_start` - opis poszczególnych pól struktury znajduje się poniżej). Drzewo AVL umożliwia wydajniejsze wykonywanie operacji, przy których potrzebne jest wyszukanie informacji o którymś z obszarów pamięci wirtualnej danego procesu.
- listę prostą jednokierunkową, również uporządkowaną względem początku obszaru. Używana w przypadku konieczności wykonania operacji na każdym obszarze pamięci procesu.

Dla obszarów związanych z danym plikiem lub segmentem pamięci dzielonej mamy listę cykliczną dwukierunkową.

Najważniejsze pola struktury `vm_area_struct`:

- `vm_mm` - wskaźnik do struktury typu `mm_struct`. Struktura ta opisuje całą pamięć wirtualną pojedynczego procesu i jest opisana w dalszej części tego rozdziału,
- `vm_start, vm_end` - odpowiednio przesunięcie początku i końca danego obszaru w przestrzeni adresowej procesu,
- `vm_page_prot` - pole określa atrybuty ochrony danego obszaru. Przyjmuje następujące wartości:
  - `PAGE_READONLY, PAGE_COPY` - do stron z tego obszaru jest możliwy dostęp tylko w trybach wykonywania lub czytania. Zapis na takiej stronie powoduje
    tt page_fault.
  - `PAGE_SHARED` - każdy rodzaj dostępu jest dozwolony.
- `vm_flags` - pole określające wszystkie własności danego obszaru, m. in. prawa do odczytu/zapisu/wykonania, informacje o tym, czy obszar jest dzielony, o możliwości zmiany praw dostępu itp.. Flagi określają również, czy obszar zawiera kod wykonywalny, oraz czy jest zablokowany.
- `vm_inode, vm_offset` - informacje dotyczące mapowanych plików.
- `vm_pte` - jeżeli obszar jest segmentem pamięci dzielonej to zawiera identyfikator tego segmentu.
- `vm_ops` - wskaźnik do struktury `vm_operations_struct` zawierającej wskaźniki do następujących funkcji obsługi:
  - `open` - funkcja używana przy dołączaniu obszaru do przestrzeni adresowej procesu,
  - `close` - funkcja używana przy usuwaniu obszaru,
  - `unmap` - funkcja używana przy 'odmapowywaniu' danych z pamięci procesu,
  - `nopage` - funkcja używana przy obsłudze błędów braku strony,
  - `swapout` - funkcja używana przy usuwaniu pamięci pomocniczej,
  - `swapin` - funkcja używana przy powrotnym wczytywaniu strony z urządzenia wymiany.

Strukturą danych opisującą całą pamięć wirtualną pojedynczego jest `mm_struct`. Dla procesu jest ona reprezentowana przez pole `mm` struktury `task_struct` opisującej proces. Struktura ta, reprezentująca układ pamięci procesu, zdefiniowana jest w pliku `include/linux/sched.h` i posiada następujące pola:

- `count` - licznik odwołań do struktury przez różne procesy,
- `pgd` - tablica katalogów stron procesu,
- `context` - aktualny kontekst procesu,
- `start_code` - początek segmentu kodu,
- `end_code` - koniec segmentu kodu,
- `start_data` - początek segmentu danych,
- `end_data` - koniec segmentu danych,
- `start_brk` - początek segmentu mapowania węzłów,
- `brk` - aktualna pozycja w segmencie,
- `start_stack` - początek stosu,
- `start_mmap` - adres początku pamięci wirtualnej procesu,
- `arg_start` - początek segmentu zawierającego parametry wywołania,
- `arg_end` - koniec segmentu parametrów wywołania,
- `env_start` - początek segmentu zmiennych środowiskowych,
- `env_end` - koniec segmentu zmiennych środowiskowych,
- `rss` - liczba ramek zajmowanych przez proces,
- `total_vm` - liczba stron zajmowanych przez proces,
- `locked_vm` - liczba zablokowanych stron procesu,
- `def_flags` - flagi - alogicznie jak przy `vm_area_struct`,
- `mmap` - wskaźnik do jednokierunkowej listy struktur `vm_area_struct` (opisujących spójne obszary pamięci), uporządkowanej w kolejności początkowych adresów obszaru,
- `mmap_avl` - wskaźnik do korzenia drzewa AVL struktur vm_area_struct
- `mmap_sem` - semafor do ochrony dostępu do struktury (przy operacjach na `mmap` i `mmap_avl`).

3.7 Przestrzeń adresowa procesu

Długość adresu liniowego wykorzystywana w procesorach x86 wynosi 32 bajty. Wynika z tego, że przestrzeń którą można zaadresować, jest ograniczona do 4 GB. Wielkość ta, nie jest w całości dostępna dla procesu. Segment jądra zajmuje 1 GB, pozostała część, czyli 3 GB, może być wykorzystana przez proces. Nie cała przestrzeń adresowa procesu traktowana jest w taki sam sposób. System Linuks oferuje programistom pewne udogodnienia. Część pamięci zawierająca kod programu może być tylko czytana, a nie można jej modyfikować. Część pamięci zawierająca dane może być zarówno czytana jak i modyfikowana.

3.7.1 Struktura segmentów pamięci procesu.

Całą pamięć dostępną dla procesu można umownie podzielić na trzy części. Pierwsza z nich, rozpoczynająca się od adresu `0xbfffffb`, zawiera nazwę pliku procesu, środowisko, argumenty z którymi wywołany był program, tablice wskaźników do argumentów i zmiennych środowiskowych oraz informacje dotyczące ładowania dynamicznego. Koniec tej części wyznacza początek stosu.

`0xbfffffb`

Nazwa pliku procesu
Środowisko
Argumenty
Informacje do ładowania dynamicznego
Wskaźniki do argumentów i zmiennych środowiskowych
Stos

Część druga zawiera kod programu, zmienne inicjalizowane w programie. Więcej informacji o tym gdzie znajdują się zmienne dynamiczne programu można znaleźć w części o wywołaniu systemowym brk().

BSS
Dane - zmienne inicjowane
Kod programu
Nagłówek, segmenty pomocnicze

`0x84000000`

Część trzecia zawiera współdzielone przez różne procesy biblioteki w C. Podobnie jak część pamięci wykorzystywaną przez program właściwy można ją podzielić na część kodu, zmiennych inicjowanych oraz zmiennych dynamicznych.

3.7.2 Wywołanie systemowe `fork()`

Wywołanie systemowe `fork()` powoduje powstanie nowego procesu i oczywiście przydzielenie mu pamięci. Nowy proces dziedziczy strukturę pamięci procesu macierzystego. Jednak nie zawsze

BSS
Dane
Kod
Nagłówek, segmenty pomocnicze

celem fork() jest skopiowanie całej pamięci procesu macierzystego dla procesu potomnego. Jeśli chcemy, aby oba procesy współdzieliły całą pamięć należy użyć wywołania systemowego `clone()`, a powstały proces nazywany jest wątkiem. W przeciwnym przypadku wykorzystujemy wywołanie systemowe `fork()`, a pamięć dostępna dla procesu potomnego jest kopią pamięci procesu macierzystego. Kopia ta nie jest wykonywana w sposób natychmiastowy. Na dodatek nie cała pamięć musi zostać skopiowana. W praktyce, aby nie manipulować bez potrzeby pamięcią, w Linuksie stosowana jest metoda nazywana **kopiowanie przy zapisie** (ang. `copy-on-write`). Polega ona na tym, iż oba procesy współdzielą daną stronę pamięci, aż do momentu, w którym jeden z nich będzie próbował zmienić jej zawartość. Sprzętowo wywoływany jest wtedy błąd pamięci, działanie procesu jest wstrzymywane, a jądro wykonuje kopię strony.

3.7.3 Różnice podobieństwa między `fork()` i `clone()`.

Jeśli przyjrzymy się treści wywołań systemowych `fork()` i `clone()` łatwo zauważymy, że obie funkcje korzystają z `do_fork()` i jedynie przekazują jej różne argumenty.

```
asmlinkage int sys_fork(struct pt_regs regs)
{
return do_fork(SIGCHLD, regs.esp, &regs);
}

asmlinkage int sys_clone(struct pt_regs regs)
{
unsigned long clone_flags;
unsigned long newsp;

clone_flags = regs.ebx;
newsp = regs.ecx;
if (!newsp)
newsp = regs.esp;
return do_fork(clone_flags, newsp, &regs);
}
```

3.8 Funkcja `brk`

W pamięci procesu znajduje się segment przeznaczony na tzw. zmienne nieinicjowane (BSS). Nie ma on stałej wielkości, jego rozmiar jest zmieniany dynamicznie, w zależności od potrzeb. Zmianę jego rozmiaru realizuje funkcja `brk`, której algorytm jest przedstawiony poniżej. Główną częścią algorytmu jest po prostu wywoływanie innych funkcji, przyłączających segment w przypadku powiększania przestrzeni adresowej, lub odłączających go w przypadku jej zmniejszania. Powiększenie segmentu pamięci użytkownika ma miejsce w przypadku dynamicznego przydziału pamięci (np. funkcja `malloc()` w C). Analogicznie zmniejszenie segmentu ma miejsce w przypadku zwolnienia pamięci.

Algorytm zmiany rozmiaru segmentu realizuje funkcja systemowa `sys_brk` [2], ktorej zadaniem jest rozpoznawanie kontekstu wywołania funkcji i wywoływanie odpowiednich funkcji systemowych. Najlepiej działanie tej funkcji można prześledzić na podstawie jej kodu źrodłowego (kolejność: instrukcja - pod spodem komentarz do niej).

```
asmlinkage unsigned long sys_brk(unsigned long brk)
{

/* brk jest to adres końca segmentu BSS jaki chcemy ustawić */

unsigned long rlim, retval;

unsigned long newbrk, oldbrk;

struct mm_struct *mm = current->mm;

down(&mm->mmap_sem);

/* opuszczenie semafora dostępu do struktury mm_struct */

lock_kernel();

/* zablokowanie jądra */

if (brk < mm->end_code)
goto out;

/* brk nie może znajdowac się w segmencie kodu */

newbrk = PAGE_ALIGN(brk);

/* wyrównanie wartości brk do początku strony */

oldbrk = PAGE_ALIGN(mm->brk);

/* na zmiennej oldbrk jest teraz początek strony pamięci
na której kończyła się poprzednia wartość brk */

if (oldbrk == newbrk)
goto set_brk;
```

[2] Znajduje się w module mmap.c.

```
/* nowa żądana wartość brk leży na tej samej stronie pamięci co stara,
więc nie trzeba przydzielać nowej strony, można od razu ustawić nową
wartość brk */

if (brk <= mm->brk)
{
if (!do_munmap(newbrk, oldbrk-newbrk))
goto set_brk;
goto out;
}
/* zawsze można zwolnić pamięć */

rlim = current->rlim[RLIMIT_DATA].rlim_cur;
if (rlim < RLIM_INFINITY && brk - mm->end_code > rlim)
goto out;

/* sprawdzamy czy nie przekroczyliśmy limitu pamięci */

if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
goto out;

/* sprawdzamy czy przydzielany segment nie zachodzi na inne segmenty */

if (!vm_enough_memory((newbrk-oldbrk) >> PAGE_SHIFT))
goto out;

/* sprawdzamy czy mamy wystarczająco dużo pamięci */

if (do_mmap(NULL, oldbrk, newbrk-oldbrk,
   PROT_READ|PROT_WRITE|PROT_EXEC,
   MAP_FIXED|MAP_PRIVATE, 0) != oldbrk)
goto out;

/* przydzielamy nowy obszar pamięci */

set_brk:
mm->brk = brk;

/* ustawiamy nową wartość brk */

out:
retval = mm->brk;

/* ustalamy co zwracamy */
```

```
unlock_kernel();
up(&mm->mmap_sem);

/* odblokowanie jądra i podniesienie semafora */

return retval;
}
```

Rozmiar segmentu można zmieniać samemu wywołując w programie funkcje biblioteczne `brk` lub `sbrk`. Postać funkcji:

```
int brk(void *end_data_segment);
void sbrk(ptrdiff_increment);
```

Opis działania:

Dla funkcji `brk()` wskaźnik `end_data_segment` jest wartością największego adresu wirtualnego segmentu danych procesu. Funkcja jako wynik zwraca 0 jeśli operacja zmiany rozmiaru segmentu się powiodła, w przeciwnym przypadku zwraca -1.

Argumentem funkcji sbrk() jest wartość o jaką ma sie powiększyć segment danych procesu. Funkcja zwraca wskaźnik do początku nowo przydzielonego obszaru.

3.9 Pamięc dzielona

Autor Łukasz Oleksak

3.9.1 Wstęp

Pojęcie pamięci dzielonej występuje bardzo często w kontekście mechanizmów komunikacji (IPC System V). Stanowi wtedy niezwykle szybki sposób wymiany informacji. Nie jest to jednak jedyne zastosowanie koncepcji współdzielenia pamięci. Segmenty pamięci dzielonej z powodzeniem wykorzystuje także jądro systemu Linux, głównie po to by umożliwić wykonywanie wielu instancji tego samego programu w oparciu o jeden, wspólny segment kodu. Współdzielone moga być również fragmenty segmentu danych (pod warunkiem, że nie ulegają one modyfikacji) oraz biblioteki dzielone. Opis mechanizmów stosowanych przez Linuksa przy realizacji wyżej wymienionych zadań stanowi treść niniejszego rozdziału.

Rozdział jest podzielony na dwie części tematyczne. Pierwsza z nich zajmuje się statycznym aspektem realizacji pamięci dzielonej i zawiera przede wszystkim opisy struktur i zmiennych systemowych wykorzystywanych w procesie zarządzania pamiecią dzieloną. Zadaniem drugiej części jest przedstawienie modelu pamięci dzielonej w ujęciu dynamicznym, czyli prezentacja wywołań systemowych odpowiedzialnych za prawidłowe funkcjonowanie tego modelu.

3.9.2 Aspekt statyczny realizacji pamięci dzielonej

Odmienna specyfika segmentów dzielonych i związana z nią różnica w między innymi obsłudze błędów braku strony oraz sposobie wymiany stron, przyczyniła się do powstania obszarów pamięci wirtualnej.[3] Obszary te określane są w systemie za pomocą struktury typu `struct vm_area_struct`.

[3]Powstanie obszarów pamięci wirtualnej wymogła także potrzeba mapowania plików do pamięci procesu.

Dokładny opis tej struktury znajduje się w treści rozdziału poświęconego organizacji pamięci wirtualnej. Z punktu widzenia pamięci dzielonej szczególnie ważne są pola:

- `unsigned long vm_pte` - informuje, że jest to obszar pamięci będący odwzorowaniem segmentu dzielonego o identyfikatorze przechowywanym w tym właśnie polu
- `struct vm_area_struct* vm_next_share, vm_pprev_share` - utrzymują cykliczną listę dwukierunkową obszarów pamięci wirtualnej podłączonych do danego segmentu pamięci dzielonej (obszary mogą należeć do różnych procesow).

Strukturą o nie bagatelnym znaczeniu jest również deskryptor segmentu pamięci dzielonej. Typ deskryptora przeznaczony dla jądra zdefiniowany jest w oparciu o typ tegoż deskryptora dostępny dla użytkownika. Cała konstrukcja wygląda nastepujaco:

```
struct shmid_ds {

        struct ipc_perm         shm_perm;     /* prawa dla operacji */
        int                     shm_segsz;    /* rozmiar segmentu w bajtach */
        __kernel_time_t         shm_atime;    /* czas ostatniego dołączenia */
        __kernel_time_t         shm_dtime;    /* czas ostatniego odłączenia*/
        __kernel_time_t         shm_ctime;    /* czas ostatniej zmiany */
        __kernel_ipc_pid_t      shm_cpid;     /* pid procesu twórcy */
        __kernel_ipc_pid_t      shm_lpid;   /* pid procesu, który wykonał ostatnią operację */
        unsigned short          shm_nattch;   /* liczba aktualnych dowiązań */
        unsigned short          shm_unused;   /* dla zgodności (nieużywane) */
        void                    *shm_unused2; /* ditto - używane przez DIPC */
        void                    *shm_unused3; /* nieużywane */
};

struct shmid_kernel {

        struct shmid_ds         u;
        /* pozostałe pola tylko dla jądra */
        unsigned long           shm_npages;   /* rozmiar segmentu w stronach */
        unsigned long           *shm_pages;   /* tablica stron segmentu -> SHMMAX */
        struct vm_area_struct   *attaches;    /* głowa listy dołączonych obszarów vma */
};
```

Deskryptory, z racji przechowywanych w nich informacji, stanowią podstawę działania wszystkich funkcji zaangażowanych w realizację pamięci dzielonej. Składowane są one w globalnej tablicy `shm_segs`, która jak każda porządna tablica ma ustalony rozmiar. Widocznym staje sie zatem, że nie dysponujemy dowolną liczbą deskryptorów.

W ten sposób dotarliśmy do zagadnienia związanego z limitami narzucanymi na pamięc dzieloną przez system Linux. Wartości limitów przechowuje się na wymienionych poniżej zmiennych:

- `SHMMAX` - maksymalny rozmiar segmentu podany w bajtach

- SHMMNI - maksymalna liczba segmentów w całym systemie
- SHMMIN - minimalny rozmiar segmentu
- SHMSEG - maksymalna ilość segmentów w jednym procesie.

3.9.3 Aspekt dynamiczny realizacji pamięci dzielonej

Rozdział ten opisuje najważniejsze algorytmy związane z zarządzaniem pamięcią dzieloną. Ze względu na jego większą złożoność podsekcje poświęcone poszczególnym operacjom budowane są zgodnie z następującym wzorcem - najpierw krótka charakterystyka, potem opis ważniejszych funkcji składowych, a na końcu realizowany algorytm. Ma to na celu usystematyzowanie informacji zawartych w tej części rozdziału.

Funkcja sys_shmget()

Funkcja ta służy do pozyskiwania identyfikatorów segmentów pamięci dzielonej.

- `int findkey(key_t key)` - funkcja przemierza globalną tablicę `shm_segs` w poszukiwaniu deskryptora segmentu dzielonego, w którego strukturze pole `u.shm_perm.key` odpowiada argumentowi wywołania; wartości tablicy `shm_segs` rozpoznane jako `IPC_UNUSED` są pomijane, zaś przy napotkaniu `IPC_NOID` wykonywanie `findkey` jest zawieszane na kolejce `shm_lock`, ponieważ oznacza to, że funkcja dotarła do deskryptora będącego w trakcie tworzenia; proces przeszukiwania musi zostać wstrzymany na wypadek gdyby tworzony deskryptor miał się okazać celem poszukiwań; instrukcje wznawiające znajdują się w ciele funkcji newseg stanowiąc dopełnienie mechanizmu synchronizacji dostępu do tablicy `shm_segs` między obiema funkcjami; w przypadku powodzenia zwracany jest indeks znalezionego deskryptora,a w razie porażki -1
- `int newseg(key_t key, int shmflg, int size)` - funkcja ma za zadanie utworzyć nowy segment pamięci dzielonej (właściwie jego deskryptor) i umieścić wskaźnik do niego w nieużywanym polu tablicy `shm_segs`; proces tworzenia chroniony jest przez umieszczenie w obsługiwanym polu tablicy flagi `IPC_NOID`; w razie niepowodzenia przywracana jest flaga `IPC_UNUSED` i zwalniane są wykonania findkey zawieszone na kolejce `shm_lock`; instrukcja odwieszajaca wykonywana jest także w przypadku sukcesu lecz zwracana przez funkcję wartoscią jest wtedy identyfikator utworzonego segmentu

Algorytm głównej funkcji:

```
int sys_shmget(key_t key, int size, int shmflg){

protokół zapewniający wyłączność operacji oraz sprawdzenie poprawności argumentów;
jeśli key == IPC_PRIVATE /* bezwarunkowe tworzenie nowego segmentu */
  to wywołaj newseg(key,shmflg,size)
w przeciwnym wypadku /* trzeba sprawdzić czy istnieje segment pasujący do klucza */
    jeśli findkey(key) == -1       /* nie istnieje */
      to jeśli shmflg == IPC_CREAT /* flaga zezwala na utworzenie nowego */
           to newseg(key,shmflg,size)
```

```
         w przeciwnym wypadku error -ENOENT
    w przeciwnym wypadku  /* segment już istnieje */
        jeśli włączone flagi IPC_CREAT i IPC_EXCL /* lepiej żeby nie istniał */
          to error -EEXIST
        w przeciwnym wypadku
            jeśli nie można skorzystać z tego segmentu
              to zgłoś odpowiedni błąd /* no cóż było blisko */

zwróć identyfikator segmentu /* sukces */
protokół końcowy m.in. odblokowanie jądra
};
```

Funkcja sys_shmat()

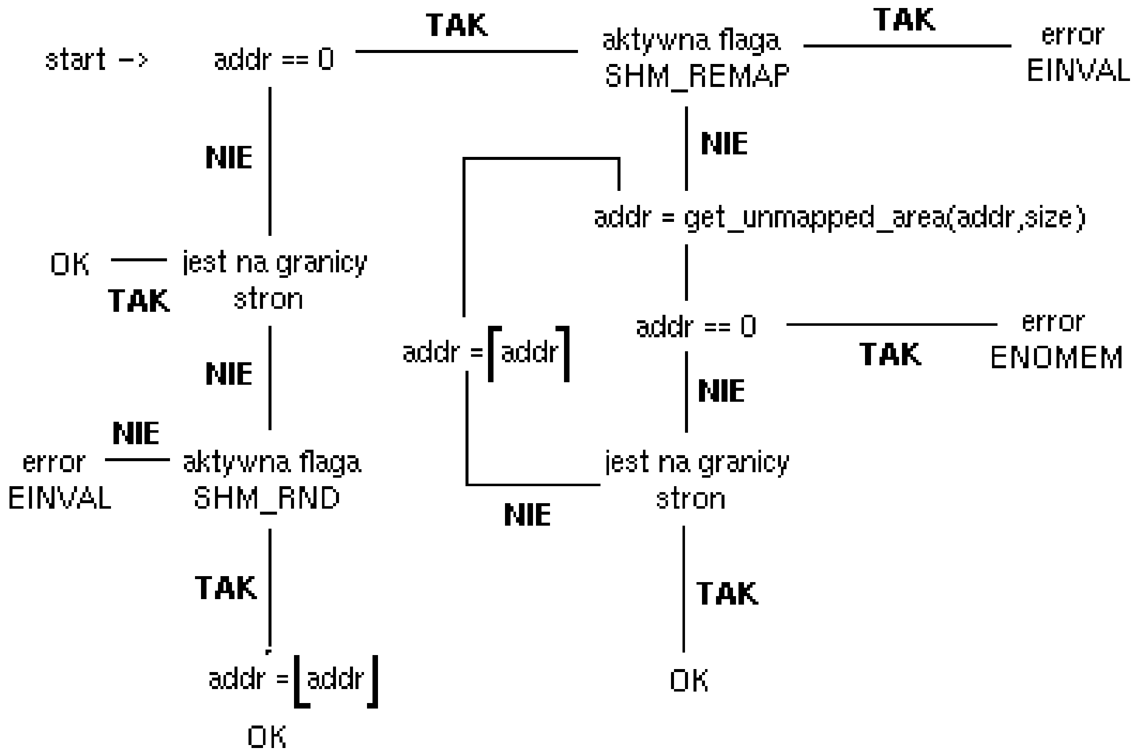
Wykorzystywana jest w celu odwzorowania segmentu dzielonego w przestrzeń pamięci wirtualnej procesu pod zadanym adresem. Wartość adresu równa 0 nakazuje systemowi aby sam wybrał adres, pod którym odwzorowana zostanie pamięc dzielona.

- `void insert_attach(struct shmid_kernel *shp, struct vm_area_struct *shmd)` - dodaje obszar pamięci wirtualnej shmd do listy attaches w deskryptorze segmentu dzielonego shp
- `int shm_map(struct vm_area_struct *shmd)` - funkcja dostaje pozytywnie zweryfikowany obszar pamięci wirtualnej, a jej zadanie polega na wpleceniu go do struktury pamięci procesu; `shm_map` czyni to za pomocą funkcji `insert_vm_struct` oraz `merge_segments`; w przypadku powodzenia funkcja zwraca 0

Algorytm głównej funkcji:

```
int sys_shmat(int shmid, char *shmaddr, int shmflg, ulong *raddr) {

blokowanie jądra;
jeśli identyfikator shmid nie wskazuje na żaden segment dzielony
  to error -EINVAL
w przeciwnym wypadku pozyskaj deskryptor wskazywanego segmentu na zmienną shp;
ustal adres pod którym ma być wmapowany segment; /* patrz rys.1 */
jeśli adres przekracza TASK_SIZE
  to error -EINVAL;
jeśli adres leży za blisko stosu
  to error -EINVAL;
jeśli aktywna jest flaga SHM_REMAP
  to jeśli obszar o początku w addr i rozmiarze shp->u.shm_segsz zachodzi
          na jakiś istniejacy obszar pamięci wirtualnej
     to error -EINVAL;
jeśli użytkownik nie ma prawa dostępu do rozpatrywanego segmentu
  to error -EACCES;
```

```
zaalokuj pamięć dla struktury obszaru pamięci wirtualnej;
jeśli nie istnieje już poprzednio ustalony segment
  to error -EIDRM;
wypełnij pola struktury obszaru pamięci;
zwiększ liczbę dowiązań do segmentu;
wywołaj shm_map;
wywołaj insert_attach;
zwróć adres na ostatnim argumencie;
};
```



Rysunek 3.1: Algorytm ustalania adresu odwzorowania.

Funkcja sys_shmdt()

Operacja odwrotna do `sys_shmat` usuwa odwzorowanie segmentu spod zadanego adresu w pamięci procesu. Wyręcza się funkcją `do_munmap`.

Funkcja shm_nopage()

Jest implementacja wariantu obsługi błędu braku strony dla segmentów z pamięcia dzieloną. Funkcja dostaje adres, do którego odwołanie wygenerowało błąd braku strony oraz zawierąjacy go obszar pamięci wirtualnej.

Algorytm:

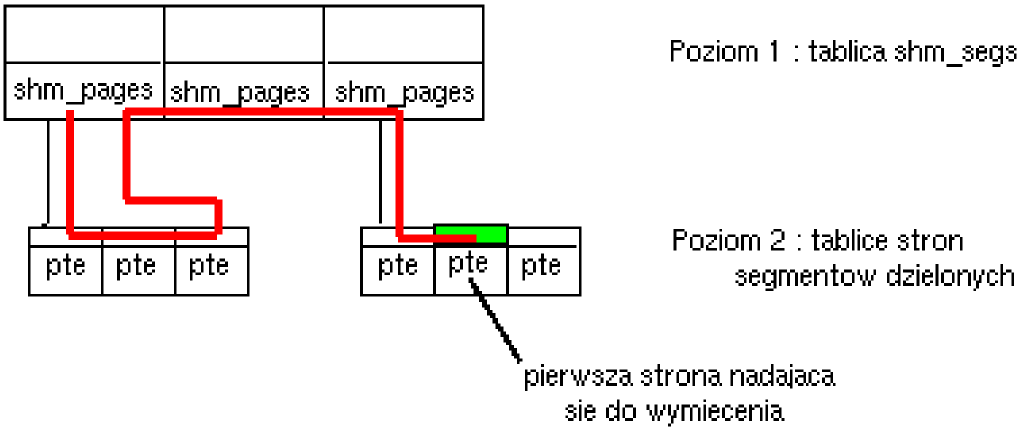
```
ulong shm_nopage(struct vm_area_struct *shmd, ulong address, int no_share){

korzystając z pola shmd->vm_pte ustal na zmiennej id identyfikator
      deskryptora segmentu dzielonego;
na zmiennej idx ustal indeks brakującej strony w tablicy shm_pages
tego deskryptora;
pobierz wartość pte spod indeksu idx;
jeśli wartość pte wskazuje, że strony wogóle nie ma
  to wykonaj page = get_free_page(GFP_USER);
     pobierz na nowo wartość pte;
     jeśli strona jest już obecna w pamięci
       to wywołaj free_page(page) i zakończ;
     jeśli strona jest obecna na urządzeniu wymiany
       to sprowadz ją do pamięci wywołując rw_swap_page_nocache() i
          zwolnij pamięc zajmowaną przez stronę na urzadzeniu wymiany;
zaznacz stronę jako brudną i dzieloną;
};
```

Funkcja shm_swap()

To cykliczne wymiatanie stron z pamięci operacyjnej do pamięci drugorzędnej realizowane przez demona `kswapd`. Intensywność wymiatania determinuje otrzymany na wejściu priorytet. Algorytm zapamiętuje swój stan aby umożliwić kontynuację procesu wymiatania od miejsca, w którym się ten proces wcześniej zakończył. Jest to oczywisty ukłon w stronę zasady lokalności. Pierwszym zadaniem algorytmu jest znalezienie w tablicy `shm_segs` deskryptora bez flag `IPC_UNUSED`, `IPC_NOID` oraz `SHM_LOCKED`, a następnie przeszukanie jego tablicy stron. Jeśli w `shm_pages` nie ma odpowiedniego kandydata do wymiecenia to szukany jest następny deskryptor i cała historia się powtarza. Przykładową ścieżkę algorytmu obrazuje rysunek nr.2.

Funkcja sys_shmctl()

Operacja ta jest jedną z tych, które swą uniwersalnością dorównują szwajcarskim scyzorykom. Za jej pomocą można uzyskać olbrzymią ilość informacji statystycznych na temat pamięci dzielonej. Jak również ustawić nie mniej liczną grupę parametrów. Wartym do zapamiętania zastosowaniem tej funkcji jest możliwość usuwania segmentów pamięci dzielonej. Wystarczy jako parametr wywołania podać flagę `IPC_RMID`.



Rysunek 3.2: Przykładowa scieżka algorytmu realizowanego przez shm_swap().

3.10 Zadania

1. Załóżmy, że w systemie potrzebna jest funkcja zwracająca największą ilość pamięci dzielonej, która jest aktualnie odwzorowana do pamięci wirtualnej pojedynczego procesu. Zaproponuj algorytm według, którego mogła by postępować ta funkcja. Zadanie posiada dwa warianty, z których pierwszy uwzględnia wszystkie odwzorowania (nawet dotyczące tego samego segmentu), a drugi zlicza tylko odwzorowania różnych segmentów. W zadaniu można założyć, że struktury jądra, na których działa algorytm nie zmieniają się w trakcie jego wykonania.

3.11 Katalog i tablica stron.

Autorzy:

- Marcin Dobrochowski
- Robert Gałach
- Robert Szawłowski

3.11.1 Wstęp.

Procesy działające w systemie nie mają bezpośredniego dostępu do pamięci fizycznej. Każdy z nich posiada swoją własną *wirtualną przestrzeń adresową.*

Podstawową ideą takiego podejścia było stworzenie pamięci wirtualnej wielokrotnie większej, niż pamięć fizyczna. Dzięki temu w systemie może działać wiele procesów, których łączne zapotrzebowanie na pamięć przekracza dostępne zasoby fizyczne. Mogą również być wykonywane pojedyncze zadania wymagające większej ilości pamięci.

Pamięć wirtualna procesu podzielona jest na dwa segmenty: jądra i użytkownika. W segmencie jądra znajdują się kod i dane jądra, zaś kod i dane procesu znajdują się w segmencie użytkownika.

Proces odwołuje się do pojedynczej komórki pamięci poprzez *adres wirtualny*, który składa się z dwóch części: strona i przesunięcie. Taki adres tłumaczony jest na *adres liniowy.* Dopiero odpowiednia interpretacja tego ostatniego udostępnia żądaną komórkę pamięci.

Tłumaczeniem poszczególnych adresów zajmuje się specjalna jednostka sprzętowa: MMU (Memory Management Unit). Dzięki temu jest ono szybkie i efektywne.

3.11.2 Konwersja adresu liniowego.

W linuksie zastosowane jest adresowanie trzypoziomowe. W *adresie liniowym* strona z *adresu wirtualnego* interpretowana jest jako:

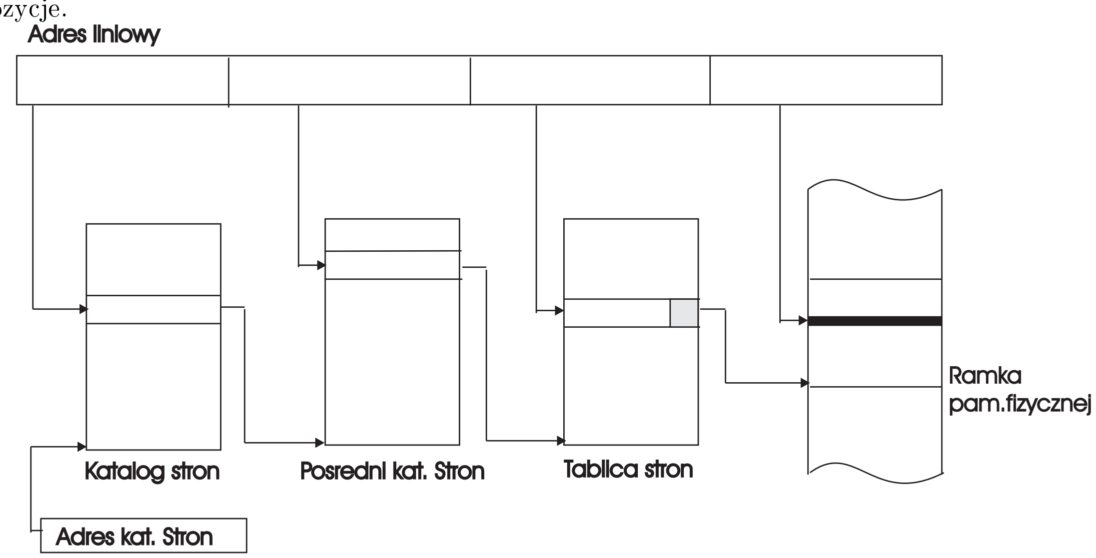
- pozycja w katalogu stron,
- pozycja w pośrednim katalogu stron
- i pozycja w tablicy stron.

Ostatnią częścią adresu liniowego jest podobnie jak w adresie logicznym przesunięcie.

W strukturze każdego procesu (*task_struct*) znajduje się pole *pgd* zawierające adres bazowy katalogu stron. Zawartość tego pola jest ładowana do odpowiedniego rejestru, gdy proces zostaje wybrany do wykonania (jest to jedna z czynności wykonywanych przy zmianie kontekstu).

Rysunek pokazuje jak wygląda przeliczanie adresu liniowego na właściwy adres fizyczny. Pierwsza część adresu wskazuje na odpowiednią pozycję w głównym katalogu stron, którego adres bazowy pobierany jest z rejestru. Zawartością tejże pozycji jest adres bazowy pośredniego katalogu stron. Druga część adresu liniowego to indeks w pośrednim katalogu stron. Pozycja w tymże zawiera adres tablicy stron. Kolejna część adresu wskazuje na pozycję w tablicy stron. Dopiero ta pozycja zawiera adres ramki pamięci fizycznej. Wówczas przesunięcie w tej ramce daje nam dostęp do żądanej komórki pamięci.

Zarówno katalogi, jak i tablice stron zajmują jedną ramkę pamięci fizycznej i posiadają po 1024 pozycje.



Rysunek 3.3: Przeliczanie adresu liniowego.

3.11.3 Konwersja adresu liniowego dla procesorów i386.

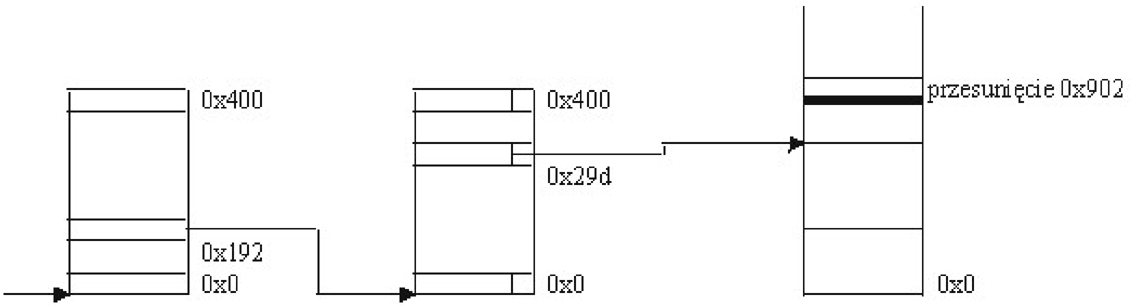
Jednak nie każda architektura sprzętu udostępnia sprzętowe wsparcie dla adresowania trzypoziomowego, które umożliwia Linuks. Bez tego niestety, konwersja adresu byłaby nieefektywna.

Wsparcia takiego nie daje m.in. architektura procesorów z rodziny Intel. Procesory te pozwalają na adresowanie dwupoziomowe, wobec czego trzeba było zastosować w systemie pewien trik. Linuks symuluje dwupoziomowe przeliczanie adresu liniowego dla procesorów Intela w następujący sposób.

Rozmiar pośredniego katalogu stron ustalony jest przez odpowiednie makro na jeden. Zaś każda pozycja w katalogu stron traktowana jest jako pośredni katalog stron. Zatem jej zawartość wskazuje na tablicę stron. Sztuczkę tę muszą uwzględniać wszystkie funkcje/makra mające dostęp do katalogu stron.

A oto przykład konwersji adresu liniowego dla procesorów Intel:
Niech adres logiczny ma postać:
64a9d902(hex) - 0110 0100 1010 1001 1101 1001 0000 0010 (bin),
wówczas mamy:

- indeks w katalogu stron, to pierwsze 10 bitów: 0110010010 (bin) - 192 (hex). Wybrana pozycja wskazuje adres bazowy tablicy stron.
- kolejne 10 bitów to indeks w tablicy stron: 1010011101 (bin) - 29d (hex). Ta pozycja może wskazywać na ramkę pamięci fizycznej;
- ostatnie 12 bitów: 100100000010 (bin) - 902 (hex) oznacza przesunięcie w ramce.



Rysunek 3.4: Konwersja adresu 64a9d902 (hex) na procesorach i386.

3.11.4 Ograniczenia.

W procesorach Intela jest adresowanie 32-bitowe. Dlatego ich architektura wspiera tylko dwupoziomowe adresowanie (po prostu takie jest wystarczające). Takie adresowanie oraz rozmiar strony pamięci ustalony na 4KB niosą za sobą ograniczenie wielkości liniowej przestrzeni adresowej procesu do 4GB.

$$1024*1024*4096\text{B} = 4\text{GB}$$

W procesorach Alpha komórki pamięci są adresowane przy użyciu 64 bitów. Dlatego zastosowanie mniej niż trzypoziomowego przeliczania adresu byłoby bardzo nieefektywne (tablice stron byłyby za duże). Dzięki zastosowaniu trzypoziomowej konwersji adresu, rozmiar przestrzeni adresowej procesu wynosi tu:

$$1023*1024*1024*8192\text{B} = 8184\text{GB}$$

Jedna pozycja w głównym katalogu stron wskazuje na niego samego (*self mapping*). Rozmiar strony to 8KB.

Pamięć wirtualna procesu jest podzielona na segment użytkownika i segment jądra. Rozmiar segmentu użytkownika na procesorach Intel ustalony jest na 3GB, a na procesorach Alpha 4TB (terabajty). Pozostała część przestrzeni adresowej przeznaczona jest na segment jądra.

3.11.5 Katalog stron.

Pozycja w katalogu stron jest czterobajtową liczbą czkowitą. Jednakże korzystanie z niej poprzez operowanie na poszczególnych bitach byłoby mało wygodne. Dlatego zostały zdefiniowane specjalne typy danych, makra i funkcje ułatwiające dostęp do tych pozycji. Wszystkie one zostały zebrane w plikach asm/page.h i asm/pgtable.h.

Pozycja w katalogu stron jest typu *pgd_t*. Dostęp do tej pozycji zrealizowany jest poprzez makro *pgd_val()*. Mamy następujące funkcje służące do operowania na *PGD*:

- funkcje *pgd_alloc()* i *pgd_free()* odpowiednio służą do przydzielania i zwalniania strony pamięci dla katalogu. Dodatkowo *pgd_alloc()* wypełnia przydzieloną stronę zerami.
- *pgd_bad()* sprawdza, czy pozyja jest poprawna,
- zaś *pgd_none()* czy pozycja została zainicjowana.
- Funkcja *pgd_clear()* usuwa pozycję z katalogu stron.
- *pgd_page()* zwraca adres strony, na którą wskazuje *PGD* (na Alphie jest to adres bazowy pośredniego katalogu stron, a na procesorach Intela - tablicy stron).
- *pgd_present()* - daje odpowiedź, czy pozycja wskazuje na pośredni katalog stron.
- *pgd_offset()* ustawia ponownie wskaźnik na pozycję w katalogu stron zawierającą adres liniowy.
- *pgd_set()* - zawartością pozycji staje się początek (adres bazowy) pośredniego katalogu stron.
- *SET_PAGE_DIR()* - to makro resetuje adres bazowy katalogu stron zadania.

3.11.6 Pośredni katalog stron.

Podobnie jak dostęp do *PGD*, zrealizowany jest dostęp do pozycji w pośrednim katalogu stron. Zaimplementowana jest struktura *pmd_t* oraz makro *pmd_val()*, poprzez które należy korzystać z tych pozycji.

Istnieją funkcje analogiczne do wymienionych w poprzednim podrozdziale. Są jednak pewne różnice:

- funkcje *pmd_alloc()* i *pmd_alloc_kernel()* przydzielają pośredniemu katalogowi stron odpowiednio stronę do zarządzania pamięcią w segmencie użytkownika lub jądra. Wszystkie pozycje inicjowane są na niepoprawne.
- Odpowiadające im funkcje *pmd_free()* oraz *pmd_free_kernel()* służące do zwalniania przydzielonej pamięci.
- Funkcja *pmd_set()* nadająca pozycji adres bazowy tablicy stron, nie jest zaimplementowana dla procesorów i386.

3.11.7 Tablica stron.

Pozycja w tablicy stron jest typu *pte_t*. Dostęp do tej pozycji, podobnie jak poprzednio realizowany jest przez makro *pte_val()*.

Pozycja taka zawiera numer ramki pamięci fizycznej. Jednakże nie zajmuje on całego miejsca w tej strukturze. Dlatego mogły się tu znaleźć różne flagi i atrybuty stron. Można korzystać z gotowych, zdefiniowanych już kombinacji tych atrybutów. Są one typu *pgprot_t*.

Atrybuty stron.

Dostępne są w Linuksie następujące makra typu *pgprot_t*:

- *PAGE_NONE* - oznacza, że pozycja w tablicy stron nie wskazuje na żadną ramkę pamięci fizycznej.
- *PAGE_SHARED* - dozwolone są wszystkie tryby dostępu do strony.
- *PAGE_COPY* - identyczne z *PAGE_READONLY*.
- *PAGE_READONLY* - oznacza że, ta strona może być odczytywana lub może być wykonywany znajdujący się na niej kod. Przy próbie zapisu następuje wyjątek. Strona może być skopiowana, a atrybut kopii ustawiony na *PAGE_SHARED*. Jest to mechanizm nazywany *kopiowaniem przy zapisie(ang. copy-on-write).*
- *PAGE_KERNEL* - dostęp do tej strony dozwolony jest tylko w trybie jądra.

Oprócz tego w *PTE* są również atrybuty ochronne strony. Możemy zdefiniować dowolną kombinację tych atrybutów, interpretowanych jako *xwr* przy użyciu makr *__P000* do *__P111* i *__S000* do *__S111* oraz makra *_PAGE_NORMAL()*. Jednak niektóre procesory nie obsługują wszystkich kombinacji. Tak jest w przypadku procesorów x86, dla których interpretacja atrybutów *xwr* jest następująca:

- atrybut w (możliwość zapisu) ma znaczenie standardowe,
- zaś jeśli którykolwiek z pozostałych atrybutów: r (możliwość odczytu) lub x (wykonywanie) jest ustawiony, interpretowane to jest tak, jakby ustawione były obydwa.

Operacje na pozycjach w tablicy stron.

Dostęp do pozycji w tablicy stron tealizowany jest przez makro *pte_val()*. Mamy do dyspozycji wiele funkcji. Oto one:

- *mk_pte()* - tworzy nową pozycję tablicy stron na podstawie adresu i zmiennej typu *pgprot_t*.
- *pte_alloc()* i *pte_alloc_kernel()* przydzielają nową tablicę stron odpowiedniow segmencie użytkownika lub jądra.
- *pte_free()* i *pte_free_kernel()* zwalniają przydzieloną pamięć.
- *pte_clear()* - czyści pozycję.
- *pte_dirty()* - sprawdza, czy włączony jest atrybut *DIRTY*.
- *pte_mkdirty()* i *pte_mkclean()* - włącza / wyłącza atrybut *DIRTY*.
- *pte_exec(), pte_read(), pte_write()* - te funkcje sprawdzają atrybuty odpowiednio: wykonywanie, czytanie i pisanie.
- *pte_mkexec(), pte_mkread(), pte_mkwrite()* - służą do włączania powyższych atrybutów.
- *pte_execprotect(), pte_rdprotect(), pte_wprotect()* - wyłączanie tych atrybutów.
- *pte_young()* - sprawdza atrybut *wiek*.
- *pte_mkold()* - włącza atrybut *wiek* - od tej pory system przyjmuje, że było odwołanie do strony.
- *pte_mkyoung()* - wyłącza ten atrybut.
- *pte_none()* - sprawdza, czy pozycja jest aktywna.
- *pte_present()* - sprawdza, czy pozycja wskazuje na ramkę pamięci fizycznej.
- *pte_page()* - zwraca adres strony wskazywanej przez pozycję w tablicy stron.
- *pte_offset()* - przenosi wskaźnik na pozycję tablicy stron oznaczającą stronę pamięci wskazywaną przez parametr. Parametrem musi być pozycja *PMD*.
- *pte_modify()* - służy do zmiany atrybutów ochrony.
- *set_pte()* - uaktywnia pozycję.

3.11.8 Literatura.

1. Zeszłoroczny projekt z Systemów Operacyjnych (wręczałki).
2. M.Beck, H.Bohme, M.Dziadzka,... „Linux kernel - jądro systemu“.
3. Źródła:

- /asm/page.h;
- /asm/pgtable.h.

3.12 Stronicowanie na żądanie.

3.12.1 Wstęp

Współczesne systemy operacyjne stoją przed problemem zwiększonego zapotrzebowania procesów na pamięć operacyjną, miedzy innymi ze względu na rozpowszechnienie się interfejsów graficznych. Rozbudowa sieci komputerowych nakłada na wielo-użytkownikowy, wielo-procesowy system sprawowanie kontroli nad zasobami pamięci jednego procesu ale takżę ochrone tychże przed innymi procesami. Stosowanym w Linux'ie rozwiązaniem tych problemów jest mechanizm stronicowania. Ochrone danych mamy zapewniona, gdyż żaden proces 'nie widzi' stron przydzielonych innemu procesowi (oprócz sytuacji gdy mamy do czynienia z pamięcią dzieloną), a więc nie może sie do nich odwoływać. Zwiększone zapotrzebowanie na pamięć, także wykraczające poza dostępną pamięć fizyczną wymusiło zastosowanie mechanizmu 'stronicowania na żądanie' polegającego na usuwaniu niepotrzebnych stron z pamięci fizycznej do pamięci pomocniczej (tu na twardy dysk). Sam mechanizm wyznaczania stron do usunięcia, ich usuwania omówiony będzie dalej (demon kswapd). Takie rozwiązanie narzuca pewne wymagania sprzętowe (możliwość przekazania systemowi kontroli nad dostępem do pamięci -wyjątek 'page fault'), ale umożliwia wieksze wykorzystanie możliwości logicznej przestrzeni adresowej i udostępnienie większej ilości pamięci niż jest dostępne fizycznie.

3.12.2 Struktury danych używane przy stronicowaniu na żądanie

Przestrzeń adresowa każdego procesu podzielona jest na obszary pamięci wirtualnej (Virtual Memory Areas), z których każdy może mieć ustawione różne atrybuty ochrony, mechanizmy obsługi błędu oraz sposoby usuwania z pamięci fizycznej -np. fragment pliku usuwany jest na innych zasadach niż inne dane procesu.

Najważniejsze pola struktury `vm_area_struct`:

- `vm_mm` - wskazuje do jakiego procesu należy dany obszar;
- `vm_start, vm_end` - przesunięcie poczatku i końca obszaru w przestrzeni adsresowej
- `vm_page_prot` - atrybuty ochrony dla danego obszaru
- `vm_flags` - pole określające własności danego obszaru: prawa odczytu/zapisu/wykonywania, czy jest to obszar pamięci dzielonej
- `vm_file, vm_offset` - jeślijest to odwzorowanie pliku to jest to wskaźnik na strukture opisujaca ten plik i przesunięcie w tym pliku

- `vm_ops` - wskaźnik do struktury zawierającej wiadomości o obsłudze różnych sytuacji dla tego obszaru: m.in. open, close, nopage, swapout, swapin

Struktury te zorganizowane są w większe całości dla jednego lub więcej procesów, mianowicie:

- w drzewo AVL uporzadkowane względem początku obszaru -szybka obsługa sytuacji krytycznych -błędu strony (osobne dla każdego procesu)
- jednokierunkową listę także uporządkowaną względem początku obszaru, używaną w przypadku, gdy jakaś operacja ma być wykonana na wszystkich obszarach pamięci procesu np. dla 'fork'-a (osobna dla każdego procesu)
- dwukierunkowa cykliczna lista oszarów związanych z danym plikiem (niekoniecznie związana z jednym plikiem)

3.12.3 Odwołanie do strony poza pamięcią

(Ten rozdział nie powstał)

3.12.4 Wymiana stron

(Ten rozdział nie powstał)

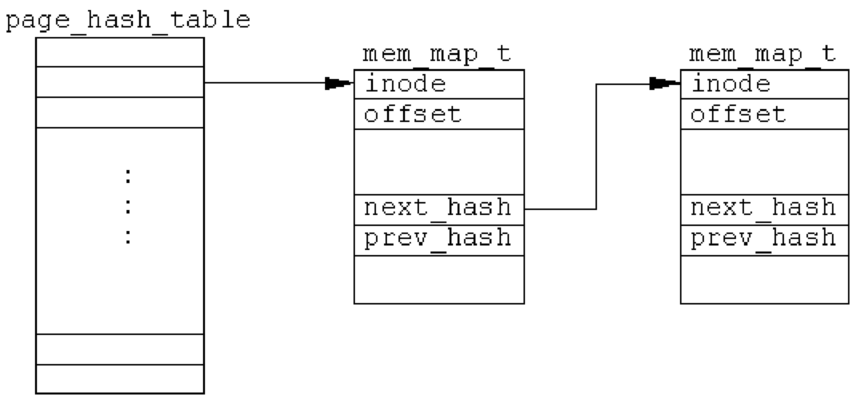
3.12.5 Podręczna pamięc buforowa stron

Mechanizm stronicowania plików pozwala nam przypisać dane zawarte w pliku dyskowym, stronom pamięci wirtualnej procesu. Strona taka nie musi znajdować się w pamięci operacyjnej, ponieważ może być do niej przywołana bezpośrednio z pliku, gdy zajdzie potrzeba operacji na danych. Ze względu na to, że dyski, na których znajdują się mapowane pliki są zdecydowanie wolniejsze od pamięci operacyjnej, w systemie Linux zastosowano podręczną pamięć buforową stron.

Rolą podręcznej pamięci buforowej stron jest przyspieszenie dostępu do plików znajdujących się na dysku. Przyspieszenie to polega na redukcji niepotrzebnych odwołań do dysku. Jedna strona mapowanego pliku jest wczytywana raz i później przechowywana w podręcznej pamięci buforowej stron. W przypadku gdy kilka procesów korzysta z tego samego pliku to użycie pamięci buforowej zaoszczędzi nam wielokrotnego wczytywania tych samych danych z dysku, a także zredukuje ilość powtarzających się danych (takich samych stron wczytanych z pliku) znajdujących się w pamięci operacyjnej.

Rysunek przedstawia strukturę podręcznej pamięci buforowej stron. Pamięć buforowa składa się z tablicy haszującej `page_hash_table`, zawierającej wskaźniki do struktur `mem_map_t`. Struktury `mem_map_t` powiązane są za pomocą pól `next_hash` i `prev_hash` w listy dwustronne. Każdy indeks tablicy haszującej zawiera wskaźnik do osobnej listy takich struktur. Każdy plik w Linuksie jest identyfikowany przez i-węzeł (`inode`), a i-węzeł jest unikalny i w pełni opisuje jeden i tylko jeden plik. Aby jednoznacznie zidentyfikowac mapowaną stronę wystarczy, że użyjemy i-węzła pliku, który mapujemy oraz offsetu danych w tym pliku, od którego zaczyna się dana strona. Indeks w tablicy stron jest obliczany przez funkcję haszującą na podstawie i-węzła i offsetu w pliku. Definicja tej tablicy znajduje się w pliku `mm/filemap.c` i jest następująca:

```
struct page * page_hash_table[PAGE_HASH_SIZE];
```



Rysunek 3.5: Struktura podręcznej pamięci buforowej stron

Stała `PAGE_HASH_SIZE` to rozmiar tablicy zdefiniowany w pliku `include/linux/pagemap.h`. W tej wersji jądra stała ta jest równa 4096.

Gdy jest potrzebne sprowadzenie żądanej strony do pamięci, to jest ona wczytywana za pośrednictwem podręcznej pamięci buforowej stron. Najpierw na podstawie i-węzła i offsetu w pliku, jednoznacznie określających stronę, obliczamy odpowiedni dla strony indeks w tablicy haszującej. Dla tego indeksu przeszukujemy listę struktur `mem_map_t` sprawdzając czy żądana strona jest już w pamięci. Jeśli strona jest obecna w podręcznej pamięci buforowej stron, to wskaźnik do struktury `mem_map_t` jest zwracany przez błąd braku strony. W przeciwnym przypadku strona musi zostać wczytana do pamięci z systemu plików. Linux alokuje fizyczną stronę i wczytuje stronę z pliku znajdującego się na dysku, a następnie wstawia odpowiednią strukrurę `mem_map_t` do bufora stron.

Jeżeli to tylko możliwe, Linux inicjuje wczytywanie następnej strony z pliku. Takie wczytywanie z wyprzedzeniem dodatkowej strony ma na celu przyspieszenie dostępu do danych z pliku jeśli proces korzysta z nich po kolei. Założenie, że dane są używane po kolei jest bardzo zasadne, ponieważ najszerszym zastosowaniem mechanizmu mapowania plików jest ładowanie przez jądro plikow wykonywalnych, a przebieg ich wykonania (czyli dostęp do kolejnych danych) jest zazwyczaj sekwencyjny.

Strony w pamięci byforowej stron są zwalniane dopiero, gdy zakończy działanie ostatni proces, który z tych stron mógłby korzystać. Gdyby jednak w systemie zabrakło pamięci fizycznej dla programów, to strony te zostaną usunięte z pamięci, pomimo, iż mogą być jeszcze potrzebne.

Podstawowe operacje dotyczące podręcznej pamięci buforowej stron zdefiniowane są w pliku `include/linux/pagemap.h`. Poniżej zamieszczona została lista funkcji dotyczących obsługi bufora oraz krótki opis ich dotyczący:

- `page_hash`(i-węzeł, offset) - funkcja haszująca, która zwraca indeks w tablicy `page_hash_table`, obliczony na podstawie wartości parametrów **i-węzeł** oraz **offset**.
- `find_page`(i-węzeł, offset) - ta funkcja poszukuje w pamięci buforowej strony o zadanym **i-węźle** oraz **offsecie**. Jeśli strona zostanie znaleziona to zostaje zwrócony jej adres (wskaźnik do struktury `mem_map_t` opisującej tę stronę), a w przeciwnym przypadku zwracana jest wartość `NULL`.

- `remove_page_from_hash_queue`(strona) - operacja ta służy do usuwania strony (**strona** jest tutaj reprezentowana przez strukturę `mem_map_t`) z pamięci buforowej.
- `add_page_to_hash_queue`(strona, i-węzeł, offset) - jest funkcją wstawiającą nową stronę (strukturę `mem_map_t`) do pamięci buforowej.

Do opisanych powyżej funkcji odwołują się funkcje zarządzające mapowaniem plików do pamięci. Stronicowanie plików jest zakodowane w pliku `mm/filemap.c`. Najważniejszą funkcją korzystającą ze stronicowania plików, znajdującą się w tym pliku jest funkcja `filemap_nopage`. Wywołuje ona funkcję `find_page` w celu sprawdzenia czy strona znajduje się już w podręcznej pamięci buforowej stron. Jeśli strona zostanie znaleziona to jej adres jest zwracany jako wynik `filemap_nopage`. Jeśli strony tej nie ma w pamięci to następuje wczytanie jej wraz ze stroną następną i umieszczenie informacji o nich, w pamięci buforowej stron, za pomocą funkcji `add_to_page_cache` (woła ona min. funkcję `add_page_to_hash_queue`). W tym przypadku jako wynik funkcji `filemap_nopage` zwracany jest adres do wczytanej przed chwilą strony. W opisie tym pominięto mniej istotne, ze względu na działanie pamięci buforowej, szczegóły dotyczące działania funkcji `filemap_nopage`.

3.13 Przydział pamięci dla jądra Linuksa w wersji 2.2.12.

Autorzy rozdziału:

- Adrian Szyndela
- Łukasz Głąb

1. Wprowadzenie
2. Schemat działania alokatora pamięci z buforowaniem obiektów.
   (a) Cache.
   (b) Tafle.
   (c) Bufory.
3. Implementacja alokatora w jądrze Linuksa w wersji 2.2.12.
   (a) Najważniejsze struktury danych.
      i. Struktura kmem_slab_s.
      ii. Struktura kmem_bufctl_s.
      iii. Struktura kmem_cache_s.
   (b) Przydział pamięci dla jądra.
   (c) Zwalnianie pamięci.
4. Uwagi końcowe.
5. Bibliografia.

3.13.1 Wprowadzenie.

Przydział pamięci dla jądra z rodziny 2.2.x uległ zasadniczej zmianie w porównaniu do poprzednich wersji. Zastosowany został alokator pamięci z buforowaniem obiektów (ang. *Object-Caching Kernel Memory Allocator).* Charakteryzuje się on lepszym wykorzystaniem pamięci (mniejsza fragmentacja) i krótszym czasem dostępu. Ogólna idea alokatora polega na wyeliminowaniu inicjalizacji i niszczenia przydzielanych obiektów (rozumianych tutaj jako obszary pamięci określonych rozmiarów). Alokator pracuje w trybie wielowątkowym (czyli potrafi zwalniać i przydzielać pamięć dla współbieżnych/równoległych procesów jądra), dlatego zastosowane są w nim semafory synchronizujące operacje na strukturach utrzymujących stan pamięci i zarządzających nią.

3.13.2 Schemat działania alokatora pamięci z buforowaniem obiektów (ang. *Object-Caching Kernel Memory Allocator*).

Buforowanie obiektów jest techniką zarządzania obiektami, które są często przydzielane i zwalniane. Pomysł polega na zapamiętywaniu stanu początkowego skonstruowanego obiektu w celu uniknięcia każdorazowej inicjalizacji. Takie buforowanie znacznie przyspiesza przydział, ze względu na większy koszt konstrukcji obiektu niż alokacji pamięci dla niego.

Cache.

Cache jest logiczną strukturą zawierającą stan wolnej i przydzielonej pamięci. Umożliwia następujące operacje:

- przydział obiektu: jeśli jest obiekt w cache'u to weź go, wpp przydziel pamięć (fizyczną) i skonstruuj obiekt;
- zwolnienie obiektu: zwróć obiekt do cache'u (bez niszczenia go);
- odzyskanie pamięci (fizycznej): weź jakieś obiekty z cache'u, zniszcz je i zwolnij pamięć (fizyczną);

Obiekty są inicjowane tylko raz, przy konstrukcji. Dzięki temu operacje przydziału i zwalniania obiektów stają się bardzo proste.

Tafle (ang. *slab*).

Tafla jest logiczną strukturą zawierającą cykliczną, dwustronną, częściowo posortowaną listę buforów tego samego rozmiaru - najpierw pełne, następnie częściowo zajęte, a na końcu puste bufory. Fizycznie jest to jedna lub kilka stron pamięci logicznie podzielonych na bloki jednakowego rozmiaru plus struktura zarządzająca taflą oraz lista struktur zarządzających buforami. Posiada licznik zajętych bloków. Z takiego schematu płyną następujące korzyści:

- trywialny algorytm zwalniania nieużywanej pamięci;
- szybkie, efektywne operacje przydziału i zwalniania pamięci;
- ograniczenie fragmentacji zewnętrznej;
- zminimalizowanie fragmentacji wewnętrznej;

Struktura tafli jest zorganizowana w zależności od wielkości buforów: mniejsze lub większe od 1/8 strony (wartość 1/8 - dobrana eksperymentalnie).

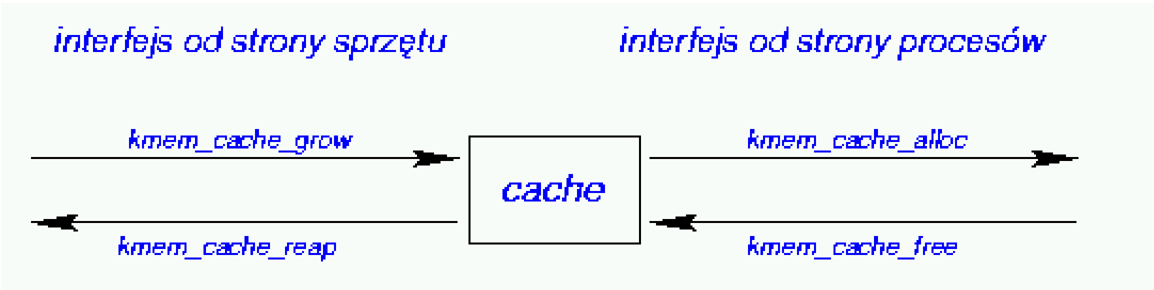
Bufory.

Struktura logiczna reprezentująca obiekty.

3.13.3 Implementacja alokatora w jądrze Linuksa w wersji 2.2.12.

Najważniejsze struktury danych.

1. **Struktura kmem_cache_s.** Struktura ta implementuje stan cache'u. Operacje na cache'u można podzielić na:
   - tworzenie/niszczenie:
     - kmem_cache_create() - tworzenie
     - kmem_cache_shrink() - niszczenie (w rzeczywistości tylko maksymalne zmniejszenie cache'a)
   - interfejs od strony procesów (moduły):
     - kmem_cache_alloc() - przydział obiektu
     - kmem_cache_free() - zwolnienie obiektu
   - interfejs od strony sprzętu
     - kmem_cache_grow() - przydział pamięci fizycznej
     - kmem_cache_reap() - zwolnienie pamięci fizycznej



Rysunek 3.6: Operacje na cache'u

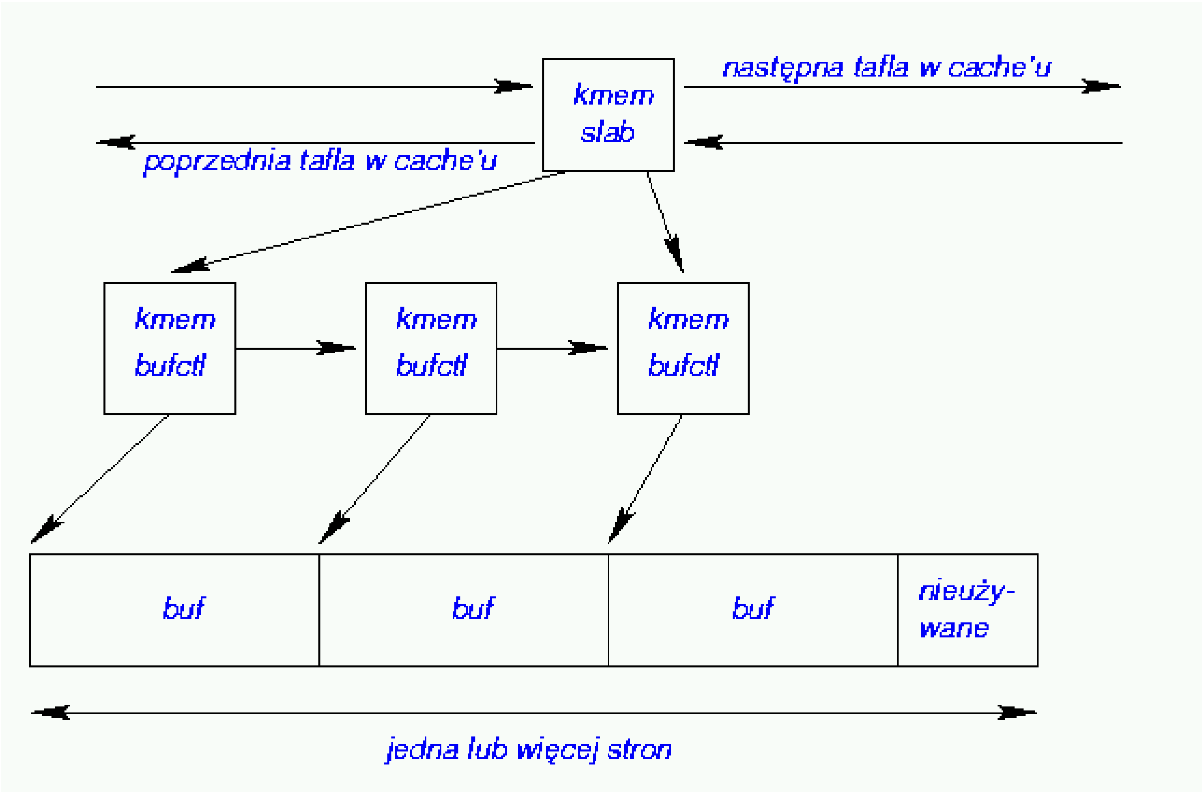
Najważniejsze pola struct kmem_cache_s:

- kmem_slab_t *c_freep - wskaźnik do pierwszej wolnej tafli;
- const char *c_name - nazwa cache'a;
- void (*c_ctor)(void *, kmem_cache_t *, unsigned long) - konstruktor;
- void (*c_dtor)(void *, kmem_cache_t *, unsigned long) - destruktor;
- struct kmem_cache_s *c_nextp - wskaźnik do następnego cache'a;

- kmem_slab_t *c_firstp, *c_lastp - wskaźniki do pierwszej i ostatniej tafli;
- spinlock_t c_spinlock - semafor;
- unsigned long c_growing - flaga: jeśli jest ustawiona, to nie można zmniejszyć ceche'a;
- unsigned long c_gfporder - 2ĉ_gfporder = liczba stron przypadających na jedną taflę;

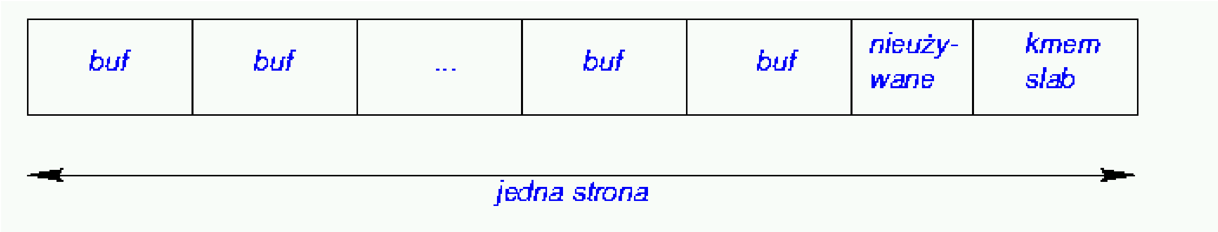
2. **Struktura kmem_slab_s.** Struktura ta implementuje stan tafli. Zawiera między innymi wskaźniki na sąsiednie tafle w cache'u (struct kmem_slab_s *s_nextp, *s_prevp), licznik zajętych buforów (unsigned long s_inuse), listę wolnych buforów (struct kmem_bufctl_s *s_freep) oraz adres pierwszego buforu w tafli (void *s_mem). Wyróżnić można dwa rodzaje organizacji tafli:

   - dla obiektów większych od 1/8 strony: każdy bufor jest zarządzany przez strukturę kmem_bufctl_s



Rysunek 3.7: Organizacja tafli dla obiektów większych od 1/8 strony

   - dla obiektów mniejszych od 1/8 strony: strona jest podzielona na jednakowego rozmiaru bufory z wydzieloną częścią zarządzającą (może powstać mały niewykorzystany fragment pamięci)



Rysunek 3.8: Organizacja tafli dla obiektów mniejszych od 1/8 strony

Pola struktury kmem_slab_s::

- struct kmem_bufctl_s *s_freep - wskaźnik do pierwszego wolnego buforu;
- struct kmem_bufctl_s *s_index - wskaźnik do listy buforów;
- unsigned long s_magic;
- unsigned long s_inuse - liczba zajętych buforów;
- struct kmem_slab_s *s_nextp, *s_prevp - wskaźniki do następnej i poprzedniej tafli w liście;
- void *s_mem - adres pierwszego buforu w tafli;
- unsigned long s_offset: SLAB_OFFSET_BITS, s_dma: 1;

3. **Struktura kmem_bufctl_s.** Służy do łączenia buforów wewnątrz tafli, identyfikacji tafli zawierającej dany bufor, a także do przechowywania adresu skojarzonego buforu (obiektu). Pola:

   - struct kmem_bufctl_s *buf_nextp - wskaźnik do następnego kmem_bufctl_s;
   - kmem_slab_t *buf_slabp - tafla buforu;
   - void *buf_objp - adres buforu;

Przydział pamięci dla jądra - kmalloc().

Zwalnianie pamięci - kfree().

3.13.4 Uwagi końcowe.

Idee tej techniki przydziału pamięci dla jądra opublikował Jeff Bonwick w artykule "The Slab Allocator: An Object-Caching Kernel Memory Allocator". Uzasadnia on w nim efektywność techniki object-cachingu na podstawie wyników testów przeprowadzonych na różnych komputerach z różnymi algorytmami przydziału pamięci dla jądra. Przeprowadzone testy sugerują, że wkrótce w większości systemów uniksowych omówiona tutaj technika będzie standardem. Linuksowa wersja przydziału pamięci dla jądra zawiera również bardzo pomocny mechanizm odpluskwiania (debugging), który umożliwia wykrywanie błędów w modyfikowanym kodzie bez wielkiego wysiłku - wystarczy włączać przy pomocy zdefiniowanych makr odpowiednie części kodu testującego. Ponadto

Algorytm 3 kmalloc()

Dane wejściowe: Rozmiar żądanego bloku pamięci
Wynik: Adres wolnego obszaru pamięci

1: Wyszukaj cache, w którym są obiekty o rozmiarach minimalnych większych od zadanego rozmiaru
2: Zablokuj przerwania (opuść semafor)
3: **jeśli** wszystkie tafle są zapełnione **to**
4:    to idź do (15)
5: **koniec jeśli**
6: **jeśli** żądanie dotyczy pamięci z dostępem bezpośrednim **to**
7:    idź do (16)
8: **koniec jeśli**
9: **jeśli** w liście tafli nie ma wskaźnika do wolnego obiektu **to**
10:    idź do (14)
11: **koniec jeśli**
12: Odblokuj przerwania (podnieś semafor)
13: Zwróć obliczony wskaźnik do wolnego obiektu
14: Zaktualizuj wskaźnik do pierwszej tafli z wolnymi obiektami
15: Dodaj nową taflę z wolnymi obiektami i idź do (3) (jeśli się to nie powiedzie, to wyjdź z błędem)
16: Zwróć taflę z obiektami w pamięci z dostępem bezpośrednim. Idź do (9)

Algorytm 4 kfree()

Dane wejściowe: Adres zwalnianego obszaru pamięci

1: **jeśli** adres zwalnianego wskaźnika jest błędny **to**
2:    idź do (7)
3: **koniec jeśli**
4: Znajdź strukturę opisującą bufor (bufctl) z uwzględnieniem miejsca położenia w pamięci (na liście s_index lub bezpośrednio za buforem - sprawdzenie flagi w c_flags).
5: Sprawdź poprawność wywołania przez sprawdzenie flagi s_magic - jeśli jest niewłaściwa, to wyjdź z błędem.
6: Zwolnij pamięć pod wskazanym adresem i zaktualizuj struktury zarządzające pamięcią.
7: Zwróć błąd i wyjdź.

jednym z parametrów funkcji tworzącej nowy cache (strukturę z obiektami) jest jego nazwa. Dzięki temu procesy mogą tworzyć cache'e identyfikowane nazwami związanymi z nazwą danego procesu - w pliku slabinfo w katalogu /proc są zapisywane te nazwy wraz z rozmiarami przydzielonej im pamięci - umożliwia to kontrolowanie wykorzystania pamięci przydzielanej dla jądra w czasie pracy systemu.

3.14 Zarządzanie plikami wymiany i urządzeniami wymiany

Autorzy:

- Piotr Herda
- Jacek Karczmarczyk

3.14.1 Rodzaje urządzeń wymiany

Mechanizm wymiany stron zakłada istnienie szybkiej i stosunkowo dużej pamięci pomocniczej służącej do 'wymiatania' na nią nie używanych stron. Implementują ją urządzenia wymiany, najszęściej urządzenia dyskowe. Mogą one być zorganizowane na dwa sposoby: plikowo i blokowo. Te pierwsze działają tak samo jak pliki,tylko stałej wielkości, czyli dane są umieszczone w blokach rozrzuconych po dysku, dostęp może trwać dłużej. Poza tym trzeba utrzymywać informację w których blokach pamiętana jest dana strona. Urządzenia blokowe nie posiadają tych wad, zajmują spójny obszar na dysku, dostęp jest szybszy.

3.14.2 Struktury danych

Informacja o podłączonych urządzeniach wymiany stron pamiętana jest w tablicy `swap_info`, o rozmiarze określonym przez stałą `MAX_SWAP_FILES`, jest o ograniczenie na liczbę urządzeń wymiany stron w systemie.

Elementy tej tablicy są typu:

```
struct swap_info_struct{
        unsigned int flags;
        kdev_t swap_device;
        struct dentry * swap_file;
        unsigned short * swap_map;
        unsigned char * swap_lockmap;
        unsigned int lowest_bit;
        unsigned int highest_bit;
        unsigned int cluster_next;
        unsignet int cluster_nr;
        int prio;
        int pages;
        unsigned long max;
        int next;
};
```

`flags`
flagi określające czy urządzenie używane i czy można pisać

- `SWP_WRITEOK` – można pisać
- `SWP_USED` – używane, ale nie można pisać
- inne – nie używane

`swap_device`
numer urządzenia blokowego
`swap_file`
wskaźnik do i-węzła urządzenia
`swap_map`
mapa bajtowa ramek urządzenia, mówi ile jest dowiązań do danej ramki
`swap_lockmap`
mapa bitowa, czy ramka jest dostępna
`lowest_bit`
indeks ramek urządzenia, poniżej którego wszystkie ramki są niedostępne
`highest_bit`
powyżej indeksu ramki są niedostępne
`cluster_next`
numer ostatnio przydzielonej ramki
`cluster_nr`
liczba związana z łączeniem przydzielanych ramek w pakiety
`prio`
priorytet urządzenia
`pages`
liczba dostępnych ramek na urządzeniu
`max`
maksymalny indeks ramek urządzenia
`next`
następny w liście priorytetowej

Podłączone urządzenia są powiązane w listę priorytetową, zgodnie z polem `prio`. Informacje o liście trzymane w strukturze `swap_list` typu

```
struct swap_list_t{
        int head;
        int next;
 };
```

`head`
głowa listy
`next`
następne urządzenie do użycia

3.14.3 Dołączanie nowego urządzenia

Do dołączania urządzenia wymiany służy funkcja
`int sys_swapon(const char * specialfile, int swap_flags)`
Jeżeli podłączenie się powiodło zwraca 0, jeżeli nie kod błędu. Funkcja podłącza urządzenie o nazwie `specialfile`, parametr `swap_flags` może określać priorytet urządzenia, jeżeli nie określa priorytet jest nadawany zgodnie z licznikiem `least_prioity`, zmniejszanym po każdym wykorzystaniu.Funkcja wykonuje czynności:

- sprawdza, czy użytkownik, który wywołał funkcję ma prawo przyłączyć
- szuka wolnego miejsca w tabeli `swap_info`
- ustawia priorytet
- sprawdza czy urządzenie jest odpowiednie
- inicjuje mapy bitową i bajtową
- ustawia flagę SWP_WRITEOK, wstawia do listy priorytetowej

Kody błędów:

- EPERM brak praw lub miejsca na podłączenie
- EBUSY już podłączone
- EINVAL niewłaściwe urządzene
- ENODEV nie ma takiego urządzenia
- ENOMEM brak pamięci na struktury opisujące urządzenie

3.14.4 Odłączanie urządzenia

Realizuje to funkcja
`int sys_swapoff(const char * specialfile)`
W razie powodzenia zwraca 0, inaczej kod błędu. Funkcja próbuje odłączyć urządzenie `specialfile`, w tym celu wykonuje czynności:

- sprawdza, czy użytkownik odłączający ma do tego prawo
- szuka urządzenia w tablicy `swap_info`
- usuwa urządzenie z listy i zaznacza, że nie można pisać na nim
- próbuje zwolnić ramki urządzenia, funkcja `try_to_unuse()` jeśli się nie powiedzie przywraca poprzednie ustawienia
- zwalnia pamięć zaalokowaną przy podłączeniu
- ustawia flage, że nie używane

Kody błędów:

- EPERM brak praw do odłączenia
- EINVAL niewłaściwe urządzenie w parametrze
- ENOMEM brak miejsca na zwolnienie ramek urządzenia

Funkcja `int try_to_unuse(unsigned int type)` działa w ten sposób, że przegląda procesy, by znaleźć te, których strony zostały zrzucone na urządzenie o indeksie `type` w tablicy `swap_info`.

3.14.5 Wybór urządzenia

Wybór urządzenia na które wymieciona zostanie strona jest realizowany przez funkcję:
`long get_swap_page(void)`

Zwraca ona liczbą w której jest zakodowany numer urządzenia wymiany i numer ramki na tym urządzeniu, w przypadku niepowodzenia zwraca 0. Szukanie wolnej ramki funkcja rozpoczyna od urządzenia wskazywanego przez `swap_list.next`. W razie niepowodzenia przeszukuje kolejne na liście urządzenia o tym samym priorytecie, a następnie rozpoczyna poszukiwania od początku listy. Kiedy drugi raz chce rozpoczynać szukanie od `swap_list.head`, to znaczy, że nie ma wolnej ramki na urządzeniach wymiany. Po znalezieniu wolnej ramki funkcja przestawia `swap_list.next` na następne urządzenie o tym samym priorytecie lub na początek listy.

3.14.6 Odczyt i zapis na urządzeniu wymiany

Zapis i odczyt ramki z dysku jest realizowany za pomocą funkcji `rw_swap_page_base(int rw, unisgned long entry, struct page * page, int wait)` wywoływanej przez funkcję `rw_swap_page`. Przed wykonaniem całej operacji sprawdzane są następujące warunki:

- poprawność numeru urządzenia (otrzymanego przez `SWP_TYPE(entry)`)

  `#define SWP_TYPE(entry) (((entry) » 1) & 0x3f)`
- poprawność numeru ramki (`SWP_OFFSET(entry)`)

  `#define SWP_OFFSET(entry) ((entry) » 8)`
- mozliwość zapisu (w przypadku zapisu)

Jeżeli testy wypadly pomyślnie zwiększamy licznik dowiązan do ramki. W przypadku operacji asynchronicznej zaznaczane są bity `PG_decr_after` oraz `PG_swap_unlock_after` w polu `flags` zmiennej `page`. Bity te oznaczają odpowiednio konieczność zmniejszenia licznika dowiązań do ramki oraz jej odblokowania po wykonaniu operacji we/wy.

```
if (!wait) {
  set_bit(PG_decr_after, &page->flags);
  atomic_int(&nr_async_pages);
}
if (PageSwapCache(page)) {
  set_bit(PG_swap_unlock_after, &page->flags);
}
set_bit(PG_free_after, &page->flags);
```

Na koniec wykonujemy już właściwą operację. Używana jest do tego celu funkcja `brw_page(int rw, struct page * page, kdev_t dev, int * b, int size, int bmap)` Po jej zakończeniu w przypadku operacji synchronicznej zmniejszamy licznik dowiązań i odblokowujemy ramkę.

3.14.7 Przydzielanie i zwalnianie miejsca na stronę

Przydzielane ramki są najpierw grupowane pakiety. Dlugość takiego pakietu wynosi standardowo 32 (stala `SWAP_CLUSTER_MAX`). System przeszukuje mapę ramek urządzenia wymiany równocześnie przydzielając wolne ramki i zliczając je. Trwa to dopóty, dopóki ilość przydzielonych ramek będzie równa dlugości pakietu. Podczas przydzielania ramek z następnego pakietu będą brane pod uwagę ramki zwolnione podczas pierwszego przebiegu.

Zwolnienie ramki polega na zmiejszeniu licznika dowiązań do ramki. Funkcja `swap_free(unsigned long entry)` najpierw sprawdza poprawność numeru urządzenia wymiany, obecność ramki na dysku oraz czy ramka jest używana. Po pozytywnym przejściu testów i zmniejszeniu licznika dowiązań aktualizowane są informacje o urządzeniu.

Ostatnim krokiem jest ustalenie nowego aktualnego urządzenia wymiany - jeśli priorytet aktualnego urządzenia jest niższy od tego, z którego zwalniana jest strona to aktualnym urządzeniem zostaje pierwsze z listy priorytetowej.

Przykład działania funckji `swap_free` (kolejno: przydział, zwolnienie i przydział ramek) jest przedstawiony na rysunku:

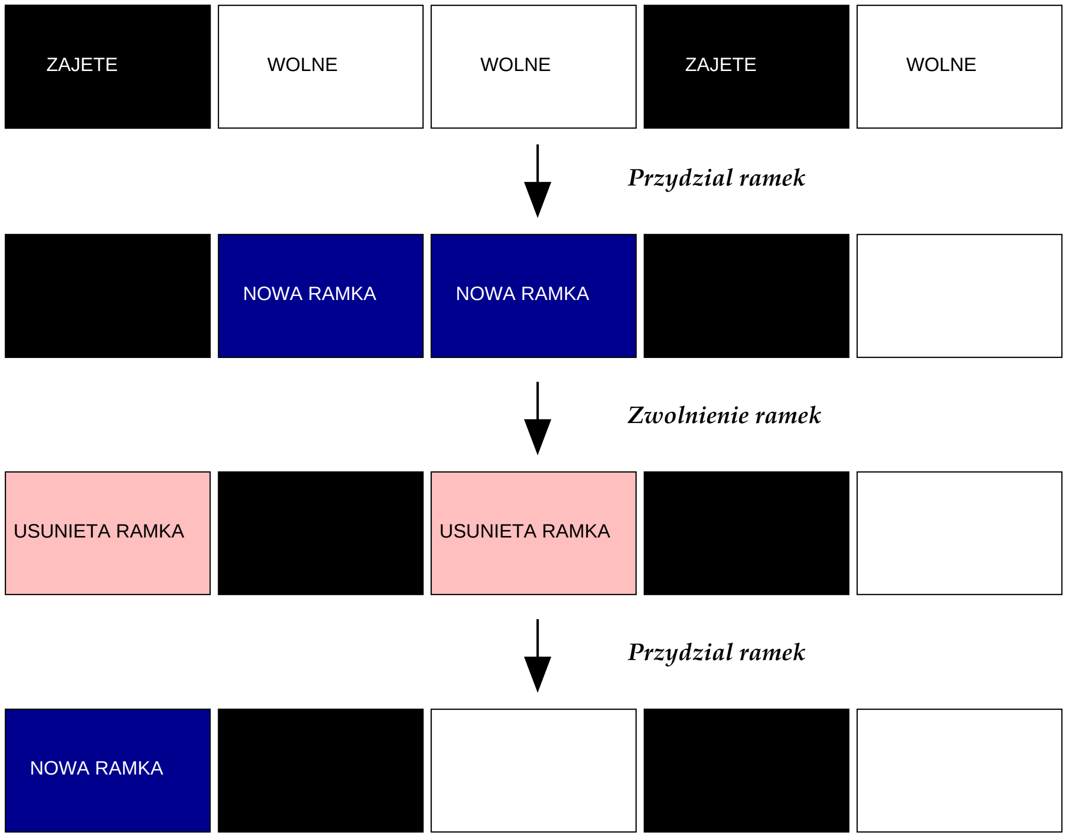
3.15 Zadania

1. Oblicz ile pamięci jest potrzebne do przechowania informacji o pomięci procesiu mającego n stron pamięci i k obszarów pamięci wirtualnej (rozmiar ramki to PS, rozmiar struktury `vm_area_struct` to $VMAS$). Załórz, że pamięć wirtualna procesu tworzy jeden obszar zaczynający się od zerowego adresu. [4]

2. Przypomnijmy algorytm wyboru procesu, z którego ma zostać wybrana strona do wymiecenia. Wartości `mm->swap_cnt` i `mm->swap_address` są zerowane przy tworzeniu procesu. W uproszczeniu do systemu Linux przyjmijmy, że strona jest wymiatana jeśli od ostatniej próby jej wymiecenia nie było do niej odwołań. Przy alokacji nowej strony, jeżeli w systemie nie ma wolnych ramek wywoływana jest funkcja `swap_out`. Zasymuluj przydzielanie stron dla dwóch procesów w systemie z 3 ramkami. Procesy wykonują:

   (a) proces 1 alokuj stronę A
   (b) proces 1 alokuj stronę B
   (c) proces 2 alokuj stronę C
   (d) proces 2 alokuj stronę D
   (e) proces 2 odwołaj się do strony C
   (f) proces 2 odwołaj się do strony D
   (g) proces 1 alokuj stronę E

[4] Piotr Sankowski



Rysunek 3.9: Przykład działania algorytmu

(h) proces 1 odwołaj się do strony E

(i) proces 1 odwołaj się do strony A

[5]

3. n-ty element katalogu stron zdefiniowy jest funkcja k(n) = 4096 * 10 * n; n-ty element i-tej tablicy stron zdefiniowany jest funkcja t(i,x) = i * 1024 + x * 4096;

   Jaki adres fizyczny otrzymamy z adres liniowego 0x001002003.

   (podpowiedz: adres liniowy 0x001002003, wskazuje na pierwszy katalog stron, druga tablice stron i offset rowny 3); [6]

4. Jaki mechanizm (lub jakie mechanizmy) powodują, że w pamięci fizycznej jest obecna co najwyżej jedna kopia kodu programu wykonywalnego dla wszystkich procesów? [7]

[5]Piotr Sankowski
[6]Marek Futrega
[7]Tomasz Fujak

Algorytm 5 swap_out

Wybierz proces `p` z największą wartością `swap_cnt`
jeśli `p->mm->swap_cnt=0` **to**
  dla każdego procesu `p` `p->mm->swap_cnt := p->mm->rss` i zacznij od początku
w przeciwnym przypadku
  dopóki Nie udało się wymieść żadnej strony **wykonuj**
    spróbuj wymieść stronę pod adresem `p->mm->swap_address`
    jeśli stronę udało się wymieść **to**
      zakończ
    koniec jeśli
    ustaw `p->mm->swap_address` na następną stronę
    jeśli nie ma więcej stron w tym procesie **to**
      ustaw `p->mm->swap_cnt` na zero
      ustaw `p->mm->swap_address` na zero
      przejdź do początku
    koniec jeśli
  koniec dopóki
koniec jeśli

5. W technice Copy On Write proces macierzysty i potomny początkowo dzielą wspólne ramki pamięci fizycznej (lub ich obrazy zapisane na urządzeniu wymiany). Jeśli jeden z tych procesów dokona zapisu do takiej wspólnej pamięci, to odpowiednia ramka zostanie skopiowana i procesowi, który dokonał zapisu jest przydzielana kopia, nawet jeżeli jest to proces macierzysty, tzn. pierwotny właściciel strony. Można by chcieć, aby proces, który posiadał stronę jako pierwszy, miał zawsze dostęp do oryginalnego jej ęgzemplarza", a kopia (kopie) byłaby przydzielana procesowi potomnemu (procesom potomnym). Odpowiedz: a) czy takie rozwiązanie byłoby możliwe/efektywne, jak możnaby to zrobić? b) czy coś byśmy na tym zyskali, lub dlaczego nic byśmy nie zyskali? [8]

6. Wyjaśnij, dlaczego w algorytmie buddy, przy zajmowaniu i zwalnianiu bloków, niektóre bity w mapach bitowych zmienia się na przeciwne. [9]

7. Mamy dany spójny obszar pamięci o rozmiarze 16 ramek (o numerach 0-15). Narysuj mapy bitowe dla następujących przypadków:
   a) Zajęte sa ramki o numerach 1,2,3,6,7
   b) Zajęte sa ramki o numerach 1,3,5,10,12,14
   c) Zajęte sa ramki o numerach 1,2,6,7,11,12 [10]

8. Mamy dany spójny obszar pamięci o rozmiarze 16 ramek (o mumerach 0-15). Zajęte są ramki o numerach: 8,13,14. Do systemu przychodzą kolejno żądania:
   I. przydziału dwóch bloków rozmiaru 2 (a i b)
   II. zwrotu ramek 8,13,14
   III. przydziału trzech bloków rozmiarów 2,2 i 1 (c,d i e)

[8] Piotr Kozbiał
[9] Anna Majkowska
[10] Anna Majkowska

IV. zwotu bloków (kolejno): d,a,b,e,c
Narysuj tablicę free_area wraz z mapami bitowymi i listami wolnych bloków w stanie początkowym oraz po każdej operacji przydziału lub zwolnienia bloku przy zastosowaniu algorytmu buddy.[11]

9. Ile obiektów można przydzielić w tafli zlokalizowanej w cache'u, w którym c_gfporder=0, a wielkość obiektu wynosi:

   (a) 204B,
   (b) 1020B

   Zakładamy, że wielkość strony wynosi 4kB.[12]

10. Ile takich samych obiektów wielkości 400 bajtów moŻna przydzielić na jedną stronę o wielkości 4096 bajtów korzystając z:

    (a) kmalloc()
    (b) kmem_cache_create() oraz kmem_cache_alloc()?

    Jaka będzie procentowo wewnętrzna fragmentacja? Można pominąć rozmiar wszelkich struktur pomocniczych.[13]

11. Jest podłączonych 5 urządzeń wymiany, połączone w listę priorytetową w kolejności: 1,2,3,4,5, o priorytetach kolejno: 8,7,7,7,6. Mają one wolnych ramek kolejno: 2,1,0,2,2. Wskaźnik `swap_list.next` pokazuje na urządzenie numer 3. Wskaźnik `swap_list.head` pokazuje oczywiście na urządzenie numer 1. Które urządzenia zostaną wybrane w czasie pięciu kolejnych wywołań funkcji `get_swap_page()` (i zrzucenia strony na wybrane urządzenie)?[14]

12. Załóżmy, że mapa bitowa urządzenia wymiany przedstawia się następująco: `0 1 0 0 1 0 1 1`, gdzie 0 - oznacza wolną pozycję na urządzeniu a 1 - zajętą. Pokaż jak będzie wyglądać sytuacja po zwolnieniu czterech, a następnie przydzieleniu czterech miejsc na strony. Operacje wykonujemy synchronicznie. [15]

13. Jak będzie wyglądać sytuacji, jeżeli najpierw przydzielimy a potem zwolnimy cztery miejsca na strony?[16]

14. Ile razy funkcja `try_to_free_page()` musi odwiedzić strony z podanymi poniżej atrybutami przed zwolnieniem zajmowanych przez nie ramek, jeżeli jest wołana z parametrem wait=0 i pomiędzy odwiedzinami żadna z tych stron nie była używana. Zakładamy, że strony nie są zarezerwowane i zablokowane.

    (a) `DIRTY=0, ACCESSED=1, AGE=0,`
    (b) `DIRTY=1, ACCESSED=0, AGE=0,`

[11] Anna Majkowska
[12] Autor zadania: Łukasz Głąb
[13] Autor zadania: Adrian Szyndela
[14] Autor: Piotr Herda.
[15] Autor: Jacek Karczmarczyk
[16] Jacek Karczmarczyk

(c) `DIRTY=1, ACCESSED=1, AGE=19,`

(d) `DIRTY=1, ACCESSED=1, AGE=2,`

(e) `DIRTY=0, ACCESSED=0, AGE=2,`

(f) `DIRTY=1, ACCESSED=0, AGE=11.`

Czy wynik byłby taki sam, jeżeli paramet wait byłby równy 1? [17]

15. W standardzie MP3 1 min utworu nagranego w stereo zajmuje około 0,9Mb. Ile czasu zajęłoby wysłuchanie utworów zajmujących całą przestrzeń adresową Linuksa. Zakładamy że, ramki mają 2 ˆ 13, a rozmiar tablicy stron wynosi 2 ˆ 12 .Ile czasu trwałoby to na komputerze z procesorem Intela 486, a ile na komputerze z procesorem ALFA ? [18]

16. Załóżmy, że w naszym systemie:

```
# define MAX_SWAPFILES 8
# define SWAP_CLUSTER_MAX 3
```

oraz wartość `least_priority` jest 10 .

(a) Wykonujemy ciąg poleceń:

```
sys_swapon ( A, swap_flags_PREFER or 101 czyli priorytet 5 )
sys_swapon ( B , bez ustalonego priorytetu )
sys_swapon ( C , swap_flags_PREFER or 1001 czyli priorytet 9 )
sys_swapon ( D, bez ustalonego priorytetu )
```

Zakładamy że: A,B,C,D są poprawnymi urządzeniami wymiany, oraz ich dołączenie przebiegło prawidłowo. Jaka jest ich kolejność na liście priorytetowej urządzeń wymiany, oraz jaki mają one priorytet? Jaka jest wartość `least_priority`?

(b) Mamy sytuację: Na liście urządzeń wymiany znajdują się cztery urządzenia o pojemności 5 ramek każde. Priorytety tych urządzeń to odpowiednio 8, 5, 5, 3. Dla ustalenia nazwijmy je A, B, C, D. Na urządzeniu A zapisanych jest pięć stron dla ustalenia nazwijmy je 1, 2, 3, 4, 5 . Na urządzeniu B zapisanych jest też pięć stron nazwijmy je 6, 7, 8, 9, 10 . Na urządzeniu C zapisane są cztery strony 11, 12, 13, 14 natomiast piąta ramka jest wolna. Na D mamy jedną stronę 15 zapisaną w pierwszej ramce. `swap_list.next` wskazuje na urządzenie o nazwie C.

A(1, 2, 3, 4, 5) $>$ B(6, 7, 8, 9, 10) $>$ C(11,12, 13, 14, ..) $>$ D(15, .. , .. , .. ,..)

Jak będzie wyglądała sytuacja (wystarczy podać gdzie zostaną zapisane nowe strony), gdy zostaną dokonane następujących operacje:

i. Z urządzeń wymiany zostanie usunięta strona 5, a następnie zostaną dodane dwie strony 16 i 17.

ii. Z urządzeń wymiany zostanie usunięta strona 6, a następnie zostaną dodane dwie strony 16 i 17.

[17] Autor Daniel Sadkowski.
[18] Autor Piotr Miłkowski.

iii. Z urządzeń wymiany zostanie usunięta strona 15, a następnie zostaną dodane dwie strony 16 i 17.

iv. Na urządzenia wymiany zostaną zapisane strony 16, 17, 18, usunięta zostanie strona 15, a następnie dodane strony 19, 20.

v. Jaki będzie efekt gdy na urządzenia wymiany zostaną zapisane strony 16, 17, a następnie przy użyciu funkcji `sys_swapoff ()` zostanie podjęta próba usunięcia urządzenia o nazwie D.

[19]

17. Czy konieczne jest opuszczanie semafora i blokowanie jądra w funkcji `sys_brk` (może wystarczy tylko opuścić semafor)? Na podstawie kodu uzasadnij odpowiedź.[20]

18. Jakie części pamięci procesu zostaną skopiowane prawdopodobnie, a jakie nie zostaną skopiowane na pewno, jeżeli użyjemy wywołania systemowego `fork` (a po nim nie nastąpi `exec`). [21]

19. Przedstaw interpretację adresu logicznego: 2e97a01c (hex), na adres fizyczny w procesorze i386.[22]

20. Na czym polega stosowany w Linuxie mechanizm 'copy on write' i jak jest realizowany?[23]

21. Jakich informacji dotyczących strony z mapowanego pliku używa podręczna pamięć buforowa stron podczas sprawdzania, czy dana strona znajduje się już w pamięci. [24]

22. Wymień operacje na strukturze danych podręcznej pamięci buforowej stron, które są wykonywane podczas wołania funkcji `filemap_nopage`. Załóż, że żądana strona nie została do tej pory jeszcze wczytana do pamięci. [25]

23. Załóżmy, że w systemie potrzebna jest funkcja zwracająca największą ilość pamięci dzielonej, która jest aktualnie odwzorowana do pamięci wirtualnej pojedynczego procesu. Zaproponuj algorytm według, którego mogła by postępować ta funkcja. Zadanie posiada dwa warianty, z których pierwszy uwzględnia wszystkie odwzorowania (nawet dotyczące tego samego segmentu), a drugi zlicza tylko odwzorowania różnych segmentów. W zadaniu można założyć, że struktury jądra, na których działa algorytm nie zmieniają się w trakcie jego wykonania.

3.16 Rozwiązania

1. Każda strona zajmuje jedną pozycję w tablicy stron, w tablicy stron przechowywana jest informacja o 1024 stronach, dla n stron potrzeba więc $\lceil \frac{n}{1024} \rceil$ ramek pamięci. W pośredniej tablicy stron przechowywana jest informacja o 1024 tablicach stron, więc na nie potrzeba

[19] Autor Ireneusz Roze.
[20] Autor Przemysław Siwik
[21] Autor Jarosław Tyszka
[22] Robert Gałach
[23] Robert Szawłowski
[24] Marcin Dobrochowski.
[25] Marcin Dobrochowski.

ramek $\lceil \frac{n}{1024^2} \rceil$. Plus jedna ramka na główny katalog stron. Ponieważ z każdym obszarem pamięci wirtualnej związana jest jedna struktura `vm_area_struct`, potrzeba więc w sumie $(\lceil \frac{n}{1024} \rceil + \lceil \frac{n}{1024^2} \rceil + 1) * PS + k * VMAS$ pamięci.

2. Prześledźmy wykonywane kroki:
   - W krokach (a,b,c) procesy otrzymują strony. W pamięci są strony ABC
   - W kroku (d) wywoływana jest funkcja wymiatająca. Ustawia ona wartości `1->mm->swap_cnt:=2`, `2->mm->swap_cnt:=1` i wybiera proces 1. Ponieważ nie było, żadnych odwołań do pierwszej strony wymiata stronę A, a `swap_address` tego procesu ustawia na stronę B. Strony w pamięci to BCD.
   - Odwołania do stron w krokach (e) i (f) nie powodują wymiecenia żadnych innych stron, gdyż strony te są w pamięci.
   - W kroku (g) wywoływana jest funkcja wymiatająca wybiera ona proces pierwszy (cały czas jest `1->mm->swap_cnt=2` i `2->mm->swap_cnt=1` mimo, że pierwszy proces ma 1 ramkę a drugi 2). Wymiatana jest strona B. Obecne w pamięci są strony CDE.
   - Odwołenie w kroku (h) jest zapamiętywane.
   - W kroku (i) odwołanie następuje do ramki, której nie ma w pamięci wykonywana jest funkcja wymiatająca, wybiera ona proces 1, gdyż nie ma w nim więcej ramek zaznacza `1->mm->swap_cnt:=0`. A następnie wybiera proces drugi, przegląda jego strony, ale ponieważ do ramek były odwołania, nie wymiata ich tylko ustawia `2->mm->swap_cnt:=0`. A następnie ponieważ dla wszystkich procesów `swap_cnt=0` to ustawia odpowiednio `1->mm->swap_cnt:=0` i `2->mm->swap_cnt:=3`. Wybiera proces drugi i usuwa stronę C. Strony w pamięci to ADE.

3. k(1) = 4096 * 10 * 1 = 40960; - to daje tablice stron
   t(k(1), 2) = 40960 * 1024 + 2 * 4096 = 41951232 - to daje adres fizyczny
   t(k(1), 2) + 3 = 41951235. - to uwzglednia offset

   Odpowiedz: 41951235.

4. Procesy powstają podczas wywołań sytemowych fork(), a nowy kod jest wczytywany podczas wykonania exec(). W przypadku funkcji fork proces potomny dziedziczy struktury związane z pamięcia; pamieć kodu nie jest kopiowana (copy-on-write kopiuje tylko pamieć do której można zapisywać). Przy wywołaniu funkcji exec pliki wykonywalne są mapowane do pamięci (do_mmap()) jeden raz miejsce (inode o tym wie, i tego pilnuje).

5. a) byłoby możliwe, ale musielibyśmy przechowywać informacje o tym do kogo dana strona pierwotnie należała (dl każdej strony), a to jednek duży koszt pamoęciowy (struktura danych) oraz czasowy (uaktualnianie struktury, na przykład przy tworzeniu lub usuwaniu procesu) b) nie moglibyśmy nic zyskać - dla systemu to tylko kłopot (patrz punkt a), a dla programów urzytkowych z założenia wszelkie mechanizmy pamięci wirtualnej, w tym stronnicowanie, maje być nie widoczne.

6. Przy wstawianiu bloków na listy i ich usuwaniu należy tak uaktualnić mapy bitowe, by były zgodne z założeniami, tzn. free_area[i].map[j]=1 wtw dokładnie jeden z bloków o numerach $2^{i+1} * j$, $2^{i+1} * j + 2^i$ znajduje się na liście bloków w free_area[i], wpp. free_area[i].map[j]=0.

7. a) free_area[0]: map=10000000 [1]: map=0100 [2]: map=00 [3]: map=1
   b) free_area[0]: map=11100111 [1]: map=0110 [2]: map=00 [3]: map=0
   c) free_area[0]: map=11000110 [1]: map=0111 [2]: map=00 [3]: map=0

8. W stanie początkowym:
   00001011<–[0]–>(9)-(12)-(15)
   0010<–[1]–>(10,11)
   00<–[2]
   1<–[3]–>(0,1,2,3,4,5,6,7)

   I. a=(10,11) b=(6,7)
   00001011<–[0]–>(9)-(12)-(15)
   0110<–[1]–>(4,5)
   10<–[2]–>(0,1,2,3)
   0<–[3]

   II. po zwrocie:
   00000000<–[0]
   0110<–[1]–>(4,5)-(8,9)
   11<–[2]–>(0,1,2,3)-(12,13,14,15)
   0<–[3]

   III. c=(4,5) d=(8,9) e=(3)
   01000000<–[0]–>(2)
   1000<–[1]–>(0,1)
   01<–[2]–>(12,13,14,15)
   0<–[3]

   IV. po zwrocie (8,9):
   01000000<–[0]–>(2)
   1010<–[1]–>(0,1)-(8,9)
   01<–[2]–>(12,13,14,15)
   0<–[3]

   po zwrocie (10-11):
   01000000<–[0]–>(2)
   1000<–[1]–>(0,1)
   00<–[2]
   1<–[3]–>(8,9,10,11,12,13,14,15)

   po zwrocie (6-7):
   01000000<–[0]–>(2)
   1100<–[1]–>(0,1)-(6,7)
   00<–[2]

1<–[3]–>(8,9,10,11,12,13,14,15)

po zwrocie (3):
0000000<–[0]
0100<–[1]–>(6,7)
00<–[2]–>(0,1,2,3)
1<–[3]–>(8,9,10,11,12,13,14,15)

po zwrocie (4-5):
00000000<–[0]
0000<–[1]
00<–[2]
0<–[3]
[4]–>(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15))

9. (a) Wielkość obiektu jest mniejsza od 1/8 wielkości strony, czyli do zarządzania jest potrzebna tylko struktura kmem_slab_s umieszczana na końcu strony, w której znajdzie się również przydzielany obiekt. Sumując wielkości pól struktury kmem_slab_s otrzymujemy: obszar = 2 * sizeof(struct kmem_bufctl_s*) + 2 * sizeof(struct kmem_slab_s*) + 3 * sizeof(unsigned long) + sizeof(void*) = 2 * 4 + 2 * 4 + 3 * 4 + 4 = 32B Obiekt zostanie przydzielony w cache'u, w którym c_gfporder=0, czyli każda tafla w tym cache'u zajmuje 1 stronę (2 ˆ c_gfporder). Strona zajmuje 4kB, więc w tafli można umieścić:

   (4096B - 32B) div 204B = 19 obiektów.

   Odpowiedź: W danym cache'u można przydzielić conajwyżej 19 obiektów.

   (b) Wielkość obiektu jest większa od 1/8 wielkości strony, czyli do zarządzania potrzebne są dwie struktury: kmem_slab_s i kmem_bufctl_s. Sumując wielkości pól otrzymujemy:

   obszar = 2 * sizeof(struct kmem_bufctl_s*) + 2 * sizeof(struct kmem_slab_s*) + 3 * sizeof(unsigned long) + sizeof(void*) + sizeof(struct kmem_bufctl_s*) + sizeof(struct kmem_slab_t*) + sizeof(void*) = 2 * 4 + 2 * 4 + 3 * 4 + 4 + 4 + 4 + 4 = 44B Podobnie jak w poprzednim punkcie każda tafla zajmuje 1 stronę, strona ma wielkość 4kB, czyli w tafli można umieścić:

   (4096 - 44B) div 1020 = 3 obiekty

   Odpowiedź: W danym cache'u można przydzielić conajwyżej 3 obiekty.

10. (a) kmalloc() - w tym przypadku aby przydzielić 400-bajtowy blok musimy skorzystać z cache'u standardowego o rozmiarach co najmniej 400 bajtów. Najmniejszym takim cache'em jest taki, który zawiera obiekty wielkości 512 bajtów. Zatem można przydzielić 8 takich obiektów. Mamy więc 112 nieużywanych bajtów na każde 400, czyli fragmentacja wewnętrzna jest wielkości 28%.

(b) kmem_cache_create() i kmem_cache_alloc() - ponieważ obiekty są takie same (czyli mają taki sam rozmiar oraz konstruktory i destruktory) możemy stworzyć cache tylko dla tych obiektów. Wtedy przydział czterystu bajtów oznacza przydział czterystu bajtów a nie jak w poprzednim przypadku pięciuset dwunastu. Czyli przydzielić można 10 obiektów tej wielkości. Zostaje nieużywane 96 bajtów na stronę, co daje 9.6 na obiekt. Mamy więc do czynienia z fragmentacją wewnętrzną o wielkości 2.4%.

11. Kolejno wybrane zostaną urzadzenia: 4,1,1,2,4.

12. Zostaną zwonione wszystkie cztery zajęte strony, a następnie przydzielone cztery miejsca od początku urządzenia: `1 1 1 1 0 0 0 0`

13. Zostaną przydzielone cztery wolne miejsca, a następnie zwolnione cztery strony licząc od początku urządzenia: `0 0 0 0 1 1 1 1`

14. (a) 5. Przy pierwszych odwiedzinach funkcja zwiększy wiek o 3, później musi 3 razy odwiedzić stronę aby jej wiek spadł do 0, i przy następnych odwiedzinach może ją usunąć.
    (b) 1. Funkcja może zwolnić pamięć przy pierwszych odwiedzinach.
    (c) 21.Przy pierwszych odwiedzinach funkcja zwiększy wiek do 20 (wartość maksymalna), a następnie jak w 1.
    (d) 7. Jak w 1.
    (e) 3. Funkcja nie zwiększa wieku strony.
    (f) 12. Jak w 5, bit DIRTY nie wpływa na liczbę odwiedzin.

    Uwaga: Paradoksalnie im strona młodsza tym jej wiek większy. Wynik byłby inny, ponieważ gdy funkcja `try_to_free_page()` wołana jest przez jądro dłużej szuka strony i może wyrzucić stonę, której parametr `AGE>0`.

15. Liczymy ile bajtów może zaadresować Linux z takimi rozmiarami ramki i tablic ramek: 2 ˆ 12 * 2 ˆ 12 * 2 ˆ 12 * 2 ˆ 13 = 2 ˆ 49 (pamiętamy o trzypoziomowym adresowaniu).

    1 min. utworu zajmuje 2 ˆ 20 * 0,9 . Dzielimy 2 ˆ 49 / (0,9 * 2 ˆ 20) = 2 ˆ 29 / 0,9 = ok. 600000000 min = 10000000 h. = ok. 417000 dni = ok. 1142 lata. Procesory Intel 486 mają adresowanie 32 bitowe (rozmiary tablic stron 2 ˆ 10, 2 ˆ 0, 2 ˆ 10, rozmiar ramki 2 ˆ 12) zatem:

    2 ˆ 32 / (2 ˆ 20 * 0.9) = ok. 75 h

    Procesory ALFA są 64 bitowe (rozmiary tablic stron 2 ˆ 10, 2 ˆ 10, 2 ˆ 10, rozmiar ramki 2 ˆ 13)zatem:

    2 ˆ 43 / 2 ˆ 20 /0.9 = ok. 17 lat

16. (a) `least_priority` = 8; kolejność B,C,D,A.
    (b) i. A(1,2,3,4,16) B(6,7,8,9,10) C(11,12,13,14,17) D(15, .. , .. , .. , ..)
        ii. A(1,2,3,4,5) B(17,7,8,9,10) C(11,12,13,14,16) D(15, .. , .. , .. , ..)
        iii. A(1,2,3,4,5) B(6,7,8,9,10) C(11,12,13,14,16) D(17, .. , .. , .. , ..)
        iv. A(1,2,3,4,5) B(6,7,8,9,10) C(11,12,13,14,16) D(.. ,17 ,18 ,19 ,20)

v. Nastąpi próba przeniesienia ramek 15 i 17, bo te będą znajdować się na urządzeniu D do pamięci. W przypadku sukcesu urządzenie zostanie odłączone. Przy porażce funkcja `sys_swapoff()` zwróci błąd `ENOMEM`.

17. Blokowanie jądra nie jest konieczne. Z treści procedury wynika, że nie ma sposobu aby coś zepsuć w trakcie zmiany rozmiaru `BSS`, ponieważ po opuszczeniu semafora żaden inny proces nie może 'grzebać' w pamięci procesu, a funkcja `sys_brk` nie korzysta z żadnych zmiennych poza lokalnymi i strukturą `mm_struct` blokowaną semaforem. Odpowiedź jest nawet udzielona w kodzie źródłowym tej funkcji.

18. Nie zostaną skopiowane następujące segmenty pamięci procesu:

    - segment kodu procesu
    - nazwa pliku procesu
    - środowisko
    - argumenty
    - część kodu segmentu współdzielonego

    Skopiowane dopiero przy próbie zapisu zostaną następujące segmenty :

    - segment danych inicjowanych
    - segment stosu
    - segment informacji do ładowania zmiennych dynamicznych
    - BSS
    - segment danych inicjowanych bibliotek współdzielonych
    - segment stosu bibliotek współdzielonych
    - BSS biobliotek współdzielonych

19. Adres logiczny dzielony jest na trzy części: pierwszych dziesięć bitów oznacza numer pozycji w katalogu stron. Następne 10 to numer pozycji w tablicy stron. Pozostałe bity to przesunięcie. Zatem najpierw należy przełożyć szesnastkowy zapis adresu na zapis binarny:

    2e97a01c (hex) = 0010 1110 1001 0111 1010 0000 0001 1100 (bin).

    Teraz grupujemy odpowiednie ilości bitów i powstałe liczby ponownie przedstawiamy w postaci skróconej:

    - numer pozycji w katalogu stron: 0010111010 (bin) = 0ba (hex),
    - numer pozycji w tablicy stron: 0101111010 (bin) = 17a (hex),
    - przesunięcie : 000000011100 (bin) = 01c (hex).

    Zatem pozycja o numerze 0ba (hex) w katalogu stron zawiera adres bazowy odpowiedniej tablicy stron. W tej tablicy pozycja o numerze 17a (hex) może wskazywać adres fizyczny ramki pamięci fizycznej i dodając do tego przesunięcie w ramce o 01c (hex) otrzymujemy żądaną pozycję pamięci.

20. Mechanizm 'copy on write' ma na celu zoptymalizowanie (minimalizacja) zużycia pamięci fizycznej. Polega na tym, że dana strona nie dostaje tak długo miejsca w pamięci, aż nie bedzie to niezbędnie konieczne, tj. nie będzie próby zapisu na niej. Możliwe to jest w dwóch przypadkach. Gdy pamięć była alokowana, ale nie było w niej jeszcze nic zapisane -strona anonimowa. Przy pierwszej próbie pisania zostanie utworzona. Drugi przypadek ma miejsce, gdy parę procesów odwołuje się do jednej strony w pamięci (np. fragment pliku), ale nie mają na niej praw do pisania. Będą wtedy tak długo korzystać z tej jednej strony, aż któryś nie będzie próbował na niej pisać. W takim przypadku tworzona jest kopia strony, uaktualniana jest struktura opisująca 'jego' strony i zostają nadane mu prawa zapisu. Mechanizm ten korzysta z flagi `VM_WRITE`.

21. Podręczna pamięć buforowa stron wykorzystuje, podczas sprawdzania obecności zadanej strony w pamięci, dwie informacje dotyczące umiejscowienia takiej strony na dysku. Pierwszą z tych informacji jest i-węzeł, mapowanego pliku, a drugą offset danych w pliku, od których rozpoczyna się strona.

22. Podczas wywołania funkcji `filemap_nopage` wykonane zostaną następujące operacje na strukturze danych podręcznej pamięci buforowej stron:

    - Zostaną przejrzane wszystkie elementy (`mem_map_t`) znajdujące się na liście stron, dla których funkcja haszująca wskazała na ten sam indeks w tablicy `page_hash_table`, co dla żądanej strony.
    - Po wczytaniu żądanej strony i ewentualnie następującej bezpośrednio po niej, zostaną dostawione struktury `mem_map_t` (opisujące nowe strony) do listy stron opisanej w poprzednim punkcie.

23. Wariant uwzględniający każde odwzorowanie.

```
longint największa_odwzorowana_ilość_pamięci_dzielonej() {

longint current, max;
int id;
struct task_struct *curr_task;
struct vm_area_struct *elmt;

dla każdej pozycji curr_task na liście procesów obecnych w systemie z wyjątkiem
           odpowiadającej procesowi init wykonaj {
    current = 0;
    dla każdego elementu elmt listy curr_task->mm.mmap wykonaj {
       /* pozyskaj identyfikator deskryptora segmentu dzielonego */
        id = SWP_OFFSET(elmt->vm_pte) & SHM_ID_MASK;
        jeśli id poprawnie identyfikuje jakiś segment dzielony
          to current += shm_segs[id]->u.shm_segsz /* dodanie rozmiaru segmentu */
        w przeciwnym wypadku błąd;
    }; /* wszystkie obszary pamięci wirtualnej procesu przejrzane */
    jeśli current > max to max = current;
}; /* wszystkie procesy uwzględnione */
```

```
zwróć max;
};
```

Literatura

[1] pliki zrodlowe jądra: ~/binfmt_elf.h

[2] Strona internetowa: http://www.cs.ucdavis.edu/~haungs/paper/node11.html

[Bonwick] Jeff Bonwick: *The Slab Allocator: An Object-Caching Kernel Memory Allocator*

[zrodla] Plik źródłowy *slab.c*