Wykład 11
Zarządzanie pamięcią
część druga: przestrzeń adresowa procesu

Pamięć wirtualna

- Idea: W danej chwili większość procesów potrzebuje tak naprawdę niewielkiego fragmentu przydzielonej przestrzeni adresowej (pewnego podzbioru stron). W związku z tym zarządzaj pamięcią tak, aby w pamięci były tylko te strony które są naprawdę potrzebne.
- Strony aktualnie niewykorzystane możemy:
  - zapisać do obszaru wymiany (np. zmodyfikowane dane).
  - czasami po prostu możemy o nich zapomnieć – dotyczy to stron tylko do odczytu, głównie kodu. Stronę z kodem zawsze możemy sprowadzić z pliku programu (lub biblioteki)
- Dzisiejszy wykład spróbuje przybliżyć jak to się dzieje w Linuksie 2.0
  - I omówić niektóre zastosowane w nim bardzo sprytne pomysły.
- **Ważna uwaga**: mechanizm pamięci wirtualnej dotyczy wyłącznie pamięci procesu.
  - Pamięć przydzielona przez jądro (__get_free_pages, kmalloc, vmalloc) nigdy nie zostanie zapisana do urządzenia.
  - Ale trzeba pamiętać, że funkcje te mogą uśpić proces wykonywujący się w trybie jądra. (Dla kmalloc, __get_free_* możemy przekazać flagę GFP_ATOMIC) aby tego uniknąć.

Sprytny pomysł jeden: współdzielenie kodu

- Uruchamiamy 30 kopii shella bash.
- Podejście naiwne: wczytujemy do pamięci RAM 30 kopii programu.
  - W związku z tym identyczny i niemodyfikowalny kod będzie powielony 30 krotnie
  - Oczywiste marnotrawienie pamięci.
- Pomysł wykorzystany w Linuksie: każda kolejna kopia “widzi” ten sam kod.
  - Pytanie: W jaki sposób zrealizowano mechanizm “widzenia”.
  - Odpowiedź: Wykorzystując stronicowanie. Nic nie stoi na przeszkodzie aby jedna strona fizyczna (ramka) była współdzielona, tzn. odwzorowana w przestrzeni adresowej kilku procesów.
  - Mechanizm współdzielenia jest wykorzystany (między innymi) do stron z kodem.
  - W tym przypadku każda ze współdzielonych stron jest widoczna dokładnie pod tym samym adresem wirtualnym, ponieważ mamy do czynienia z procesami utworzonymi z tego samego obrazu. Ale to nie musi być regułą, o czym za chwilę

Pojęcie pliku odwzorowanego w pamięci

- Standardowe wejście-wyjście z pliku (read, write) wymaga podwójnego kopiowania danych.
  - Przy odczycie dane kopiowane są z urządzenia do pamięci buforowej jądra. Następnie dane kopiowane są do pamięci procesu.
  - Przy zapisie kopiowanie następuje w przeciwną stronę.
- Jeżeli proces wczytuje duży blok danych to (a) operacja to nie jest najbardziej wydajna oraz (b) dane rezydują w dwóch miejscach w pamięci: pamięci buforowej i pamięci procesu.
- Rozwiązanie: Odwzoruj plik w pamięci. Możemy zażądać, aby plik „pojawił" się w pamięci wirtualnej procesu. Następnie operacje na tej pamięci będą operacjami na pliku.
  - Np. odczyt 10 bajtu od początku odwzorowania w pamięci jest odczytem 10 bajtu z pliku.
  - Rozwiązanie jest zaimplementowane przy pomocy mechanizmu stronicowania i ładowania na żądanie.
  - Tworzona jest tablica stron, ale bez przydziału ramek.
  - Przy próbie odczytu wyjątek błędu strony powoduje wczytanie odpowiedniej strony (często strona == 4 bufory) z pliku.
  - Przy zapisie może zajść konieczność wcześniejszego wczytania stron.

Funkcja systemowa i algorytm mmap

- void * mmap(void *addr,int length, int flags,int protection, int file, int offset).
- Funkcja ta mapuje pod adresem addr length bajtów z pliku file, począwszy od bajtu zaczynającego się od offset. (Offset musi być wielokrotnością rozmiaru strony).
- protecion to logiczne or (PROT_WRITE, PROT_READ, PROT_EXECUTE).
- Addr może być równy 0. Jeżeli jest różny od zera to jest to wskazówka (nie obowiązkowa) odnośnie adresu, pod który należy zamapować obszar pliku. Funkcja zwraca adres, pod który odwzorowała plik, lub -1 w przypadku błędu.
- Odwzorowanie obowiązuje także po zamknięciu pliku, do momentu wywołania funkcji munmap (void *addr, int length).
- Jako flagi możemy m.in. podać (nie w Linuksie 2.0.x) MAP_ANON co oznacza "odwzorowanie pliku bez podawania pliku". Jest to po prostu żądanie alokacji pamięci. (file nie ma znaczenia).
  - Implementacja malloc w bibliotece C używa mmap do alokacji dużych bloków danych (rozmiar zaokrąglany do rozmiaru strony)
  - Dla małych bloków wykorzystywany jest brk.
- Algorytm mmap nie ***ładuje żadnych danych pliku do pamięci***, tworzy natomiast tablice stron, tak aby w momencie odwołania do strony dane zostały załadowane.

Odwzorowanie prywatne i współdzielone

- Typy zadawane przy pomocy parametru flags.
- Odwzorowanie współdzielone (MAP_SHARED). Jeżeli kilka procesów odwzorowało ten sam plik, to (a) zmiany w pamięci wykonane przez jeden z nich są widoczne dla pozostałych (b) zmiany powodują zmiany w odwzorowanym pliku.
- Odwzorowanie prywatne (MAP_PRIVATE). Jeżeli kilka procesów odwzorowało ten sam pliki to (a) zmiany w pamięci wykonane przez jeden proces nie są widoczne dla pozostałych.
- W przypadku gdy nie ma prawa do zapisu odwzorowanie prywatne jest równoważne współdzielonemu. (Linuks implementuje odwzorowanie prywatne).
  - Strony są zamarkowane jako chronione przed zapisem a próba zapisu powoduje utworzenie prywatnej kopii strony.
- Odwzorowanie współdzielone jest dobrym sposobem na współdzielenie pamięci przez kilka procesów.
  - Współdzielony plik pełni rolę obszaru wymiany (swap)

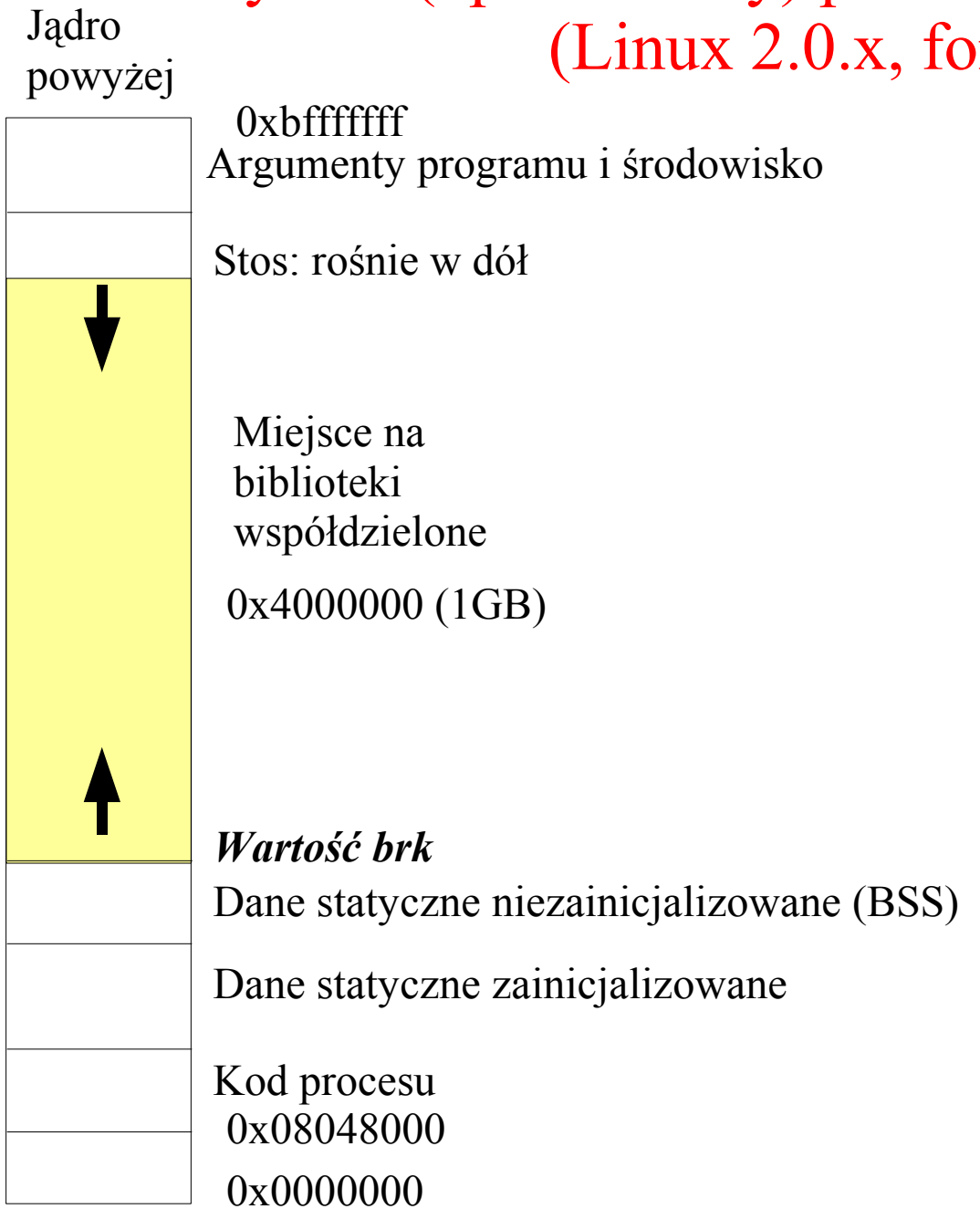
Sprytny pomysł dwa: biblioteki współdzielone (ang. shared)

- Współdzielenie kodu pomaga ale nie rozwiązuje wszystkich problemów.
- Tradycyjnie kod programu był sumą kodu przeznaczonego dla tego programu (np. skompilowane pliki .o) i kodu bibliotek (pliki .a). Linker (polecenie ld) łączył ze sobą kod i biblioteki w jeden plik wykonywalny.
- Konsekwencje: Biblioteka języka C zajmuje 1MB, przyjmijmy że w systemie jest 10 procesów,a na dysku 1000 programów.
  - Na pewno 1GB miejsca zmarnowanego na dysku.
  - Potencjalnie 10MB zmarnowanej pamięci (potencjalnie bo kolejny sprytny pomysł, który za chwilę omówię sprawia, że proces niekoniecznie rezerwuje od razu całą pamięć).
- Rozwiązanie: biblioteki współdzielone (.so – skrót od shared object). Biblioteka jest ładowana (ściślej odwzorowana w przestrzeń procesu) w chwili uruchomienia programu, a nie jego kompilacji.
  - Kod biblioteki jest współdzielony przez wszystkie procesy z niej korzystające.
  - Dane biblioteki nie są współdzielone – każdy proces ma prywatną kopię

Sprytny pomysł trzy: Ładowanie na żądanie (ang. demand loading)

- Poprzednio mówiłem o “ładowaniu” programu bądź też biblioteki. Tymczasem ładowanie całości nie ma wielkiego sensu.
  - Program może (prawie) nigdy nie wykorzystywać części kodu programu bądź też danych.
  - W przypadku biblioteki sprawa jest również istotna (chociaż i tak zyskujemy na współdzieleniu).
- W Linuksie “załadowanie” czegokolwiek oznacza utworzenie (między innymi) odpowiednich pozycji w tablicy stron. Strony na początku mogą być oflagowane jako nieobecne w pamięci.
  - Stwierdziłem mogą, ponieważ możemy mieć do czynienia z kolejnym załadowaniem obrazu programu bądź też biblioteki i strony te mogłyby zostać sprowadzone przez działające do tej pory procesy.
- W przypadku, gdy proces usiłuje się odwołać do takiej “załadowanej na niby strony” generowany jest wyjątek braku strony i dopiero wtedy jest ona sprowadzana do pamięci.
- W ten sposób strony, których nigdy nie wykorzystamy nie zostaną załadowane.
- Powyższy opis jest podobny do algorytmu mmap. Nie jest to przypadek, ponieważ załadowanie polega na wykonaniu algorytmu mmap (co najmniej dwa razy dla sekcji kodu i sekcji danych). Wykorzystywane jest odwzorowanie prywatne.

Przykład (uproszczony) przestrzeni adresowej procesu (Linux 2.0.x, format ELF)



- Dane statyczne zainicjalizowane ładowane są (na żądanie) z pliku wykonywalnego.
- Dane niezainicjalizowane są zerowane (rozmiar sekcji BSS) jest przechowywany w pliku wykonywalnym.
- Jeżeli proces potrzebuje dodatkowej pamięci, może zwiększyć swoją wartość brk.
- Biblioteki począwszy od 1GB
- Biblioteki mają również sekcje kodu, danych zainicjalizowanych oraz BSS.

Stronicowanie na żądanie

- Idea ładowania na żądanie jest rozszerzona na inne sytuacje w których nie mamy do czynienia z ładowaniem. Mówimy wtedy na stronicowaniu na żądanie (ang. demand paging) Przykłady:
  - Przy wzroście rozmiaru stosu, po przekroczeniu przez wskaźnik stosu granicy strony i wystąpieniu błędu, alokowane są kolejne strony.
  - Dane niezaiincjalizowane: przy próbie odwołania się do strony jest ona przydzielana i zerowana.
  - Zmiana wartości brk. Proces może przydzielić sobie dodatkową pamięć na dane zwiększając wartość brk. Służą do tego funkcje systemowe z grupy brk. Oczywiście zwiększenie przydzielonej pamięci nie wiąże się z zarezerwowaniem strony. Strona jest rezerwowana dopiero po próbie odwołania się do przydzielonej pamięci. (Linuks zakłada, że proces może przydzielić pamięć i nigdy z niej nie wykorzystać).
- Tak naprawdę w dwóch poprzednich przypadkach Linuks odwzorowuje pamięć w tzw. Stronę zerową (ZERO_PAGE). Jest to ramka pamięci, zabezpieczona przed zapisem, zawierająca same zera. Przydział nowej ramki następuje dopiero przy próbie zapisu. Jest to szczególny przypadek kopiowanie przy zapisie (copy on write)
- Dygresja: Linuks “nie wie” o funkcjach malloc, operatorze new etc. Te mechanizmy implementowane są na poziomie procesu, konkretnie przez bibliotekę języka C (lub C++). Oczywiście implementacja malloc wykorzystuje funkcję brk.
  - I musi utrzymywać struktury danych niezbędne do zarządzania wolnymi obszarami pamięci.

Jeszcze o bibliotekach współdzielonych

- W nagłówku pliku wykonywalnego zawarta jest lista wykorzystywanych bibliotek współdzielonych (poleceniem ldd plik_wykonywalny wyświetlisz tę listę) oraz nazwa programu (ld.so) służącego do "ładowania" tych bibliotek.
  - Jeżeli z systemu "zniknie" ld.so lub któraś ważna biblioteka (np. libc.so – biblioteka języka C) wówczas czekają nas ***olbrzymie*** kłopoty.
- Z poprzedniego rysunku wynika, że kod procesu ładowany jest zawsze pod adres x08048000. Ale pod jaki adres ładować biblioteki.
  - W systemie są setki bibliotek i jeżeli adres byłby ustalony na stałe, to prawie na pewno doszłoby do konfliktów.
- W formacie ELF (następca a.out) problem rozwiązano w ten sposób że kod biblioteki musi być działać po załadowaniu pod dowolny adres. Taki kod nazywamy PIC (ang. Position Independent Code). Generowany jest on po podaniu opcji -fpic dla kompilatora gcc.
- Niestety nie ma nic za darmo - wykorzystanie kodu PIC wiąże się ze zmniejszeniem wydajności o 5-10%. Między innymi trzeba poświęcić jeden rejestr procesora. Dlatego też nie należy umieszczać w bibliotekach dynamicznych funkcji realizujących ważne obliczenia.

Dygresja: dostęp do bibliotek współdzielonych przy pomocy API dl

API pozwalające procesom użytkownika na korzystanie bibliotek współdzielonych w przypadku gdy nie zostały jawnie zlinkowane z programem. Wykorzystywane np. do implementacji *wtyczek*. (ang. plugin).

```
#include <dlfcn.h>

void *dlopen (const char *filename, int flag);
const char *dlerror(void);
void *dlsym(void *handle, char *symbol);
int dlclose (void *handle);
```

- Program trzeba zlinkować z biblioteką dl (-ldl)
- dlopen otwiera bibliotekę. Parametr flag może być albo RTLD_LAZY (wiąż niezdefiniowane symbole w momencie wykonania kodu biblioteki albo RTLD_NOW (wiąż w momencie ładowania, każdy niezdefiniowany symbol generuje błąd).
- Wynikiem dlopen jest uchwyt, którym posługujemy się wywołując funkcję dlsym zwracającą adres symbolu (zmiennej niestatycznej lub funkcji z biblioteki) o nazwie symbol. Wynik należy zrzutować na wskaźnik do zmiennej lub funkcji.
- W wypadku błędu w dlsym lub dlopen dlerror zwróci komunikat o błędzie lub NULL gdy błędu nie było.
- Poprzez dlclose zwalniamy uchwyt i bibliotekę.

Problemy z funkcją fork

- Pierwsze wersje Uniksa używały funkcji fork do utworzenia kopii procesu. Wiązało się to (a) z przydzieleniem nowej pamięci potomkowi oraz (b) wykonaniu kopii z pamięci rodzica do pamięci potomka.
- Rozwiązanie to było niewydajne ponieważ bardzo często po wywoływaniu funkcji fork była wywoływana funkcja exec (Zwalniająca całą pamięć i uruchamiająca nowy proces). W Uniksie jeżeli inny proces chce uruchomić potomka i na niego zaczekać to wykonuje następujący kod:

```
pid = fork();
if (pid < 0) { /* Błąd !!! */
        fprintf(stderr, "Fork Failed");
        exit(-1);
}
 else if (pid == 0) { /* proces potomny */
         execlp("/bin/ls","ls",NULL);
}
else { /* proces rodzicielski */
wait(NULL);
printf("Child Complete");
```

- W klasycznym Uniksie cała pamięć zostanie skopiowana (fork) a za chwilę zwolniona (exec)

Sprytny pomysł cztery: kopiowanie przy zapisie (ang. copy on write – COW)

- Częściowe rozwiązanie problemu już znamy możemy bezpiecznie dzielić kod pomiędzy proces rodzicielski i potomka. Czy musimy kopiować strony z danymi ? Linuks robi to tylko wtedy gdy nie ma wyjścia !!!
- Po wykonaniu funkcji fork strony z danymi oflagowane są jako współdzielone (przez rodzica i potomka) oraz (uwaga !!!) zabezpieczone przed zapisem.
- Jeżeli obydwa procesy czytają strony nic się nie dzieje.
- Jeżeli jeden z procesów próbuje zmodyfikować stronę z danymi to generowany jest błąd ochrony (strona była zabezpieczona przed zapisem). Kopia strony wykonywana jest w tym momencie.
- Mechanizm COW jest wykorzystywany:
  - w wywołaniu systemowym fork
  - przy prywatnym odwzorowaniu plików – w szczególności dane programów i bibliotek.
- Powyższe rozwiązanie (i pozostałe sprytne pomysły) pozwalają na zaoszczędzenie pamięci, ale potencjalnie kosztem np. zwiększenia czasu reakcji procesów interakcyjnych, zwłaszcza w przypadku znacznego obciążenia systemu.

Jak to zaimplementować

- Te wszystkie rozwiązania pomagają (lub nie) ale ich zaimplementowanie staje się coraz trudniejsze.
- Wyobraźmy sobie wyjątek stronicowania (page fault). Jakie mogą być przyczyny:
  - Proces odwołał się do strony do której nie miał prawa się odwołać => SIGSEGV
  - Proces odwołał się do strony, którą trzeba sprowadzić (ale skąd, znowu kilka możliwości np. dane z pliku albo ze swapu, kod z pliku (jakiego)).
  - Proces się odwołał do strony którą trzeba mu przydzielić i wyzerować (np. Dane niezainicjalizowane, powiększony obszar przez BRK).
  - Proces odwołał się do strony, którą należy skopiować na żądanie.
  - Wydaje się że proces odwołał się do strony, której nie miał prawa się odwołać, ale tak naprawdę jest to wynikiem sukcesywnego powiększania stosu => stronę trzeba przydzielić.
- Widać że kod zarządzający przestrzenią adresową procesu staje się nieczytelny (masa instrukcji switch).
- Z pomocą przychodzi nam, podobnie jak w przypadku VFS, programowanie obiektowe

Pojęcie wirtualnego obszaru pamięci

- Do tej pory jedyną strukturą danych była tablica stron. Jednakże sama tablica nie jest w stanie zagwarantować sprawnej implementacji powyższych mechanizmów.
- Jest wyraźnie widoczne, że w przestrzeni adresowej mamy do czynienia z obszarami (grupami stron zajmującymi ciągły fragment przestrzeni adresowej) charakteryzującymi się podobnymi własnościami, takimi jak:
  - Identyczny sposób reakcji wyjątek stronicowania.
  - Prawa odczytu zapisu i wykonania.
- Oczywiście strony w takim obszarze mogą być umieszczone w różnych miejscach pamięci fizycznej, na dysku, albo mogą nie istnieć nigdzie (typu on demand - zerowane)
- Z powyższych przyczyn w Linuksie wprowadzono strukturę (obiekt ???) obszaru pamięci wirtualnej (struct vm_area_struct), która także zawiera wskaźnik do tablicy metod (vm_operations_struct)

Struktura mm_struct (sched.h)

```
struct mm_struct {
   int count; // Liczba użytkowników
   pgd_t * pgd; // Katalog tablicy stron
   unsigned long context;
   // Początek i koniec kodu i danych.
   unsigned long start_code, end_code, start_data, end_data;
   // Podobnie w przypadku obszru brk i stosu
   unsigned long start_brk, brk, start_stack, start_mmap;
   // Podobnie w przypadku argumentów i środowiska
   unsigned long arg_start, arg_end, env_start, env_end;
   // Informacje statystyczne – nie do końca prawdziwe
   unsigned long rss, total_vm, locked_vm;
   unsigned long def_flags;
   // Początek lista wirtualnych obszarów pamięci
   struct vm_area_struct * mmap;
   // Korzeń drzewa AVL listy wirtualnych obszarów pamięci
   struct vm_area_struct * mmap_avl;
   struct semaphore mmap_sem;
};
```

- Reprezentuje przestrzeń adresową procesu. Wskazuje na nią pole mm w strukturze task_struct.
- Drzewo AVL jest niezbędne do szybkiego przeprowadzenia operacji typu: znajdź obszar którego dotyczy dany adres, np. w funkcji verify_area sprawdzającej poprawność adresu

Współdzielenie struktur mm

- Linux, oprócz klasycznego wywołania fork, zawiera inne wywołanie systemowe o nazwie clone().
- W wyniku tego wywołania proces potomny współdzieli wirtualną przestrzeń adresową (czyli strukturę mm_struct) z procesem rodzicielskim. Stąd pole count w tej strukturze.
- W takiej sytuacji jakiekolwiek modyfikacje danych w jednym z procesów są widoczne natychmiast w drugim.
- Funkcja clone pozwala na zaimplementowanie wątków (po to zresztą powstała), np. Poprzez bibliotekę LinuxThreads implementującą wątki POSIX.
  - W Linuksie nie odróżnia wątków od procesów, wątek to proces współdzielący przestrzeń adresową z innymi wątkami.
  - Niestety takie proste rozwiązanie obniża wydajność i nie jest w 100% komatybilne ze standardem POSIX (sygnały).
- W najnowszej wersji jądra (2.6) i wersjach wcześniejszych (firmy RedHat) zastosowano nowsze i dużo lepsze rozwiązanie (NGPTL – Next Generation Posix Thread Library)

Struktura vm_area_struct

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        // początek i koniec
        unsigned long vm_start;
        unsigned long vm_end;
        pgprot_t vm_page_prot;
        unsigned short vm_flags;
        // Drzewo AVL i lista liniowa
        short vm_avl_height;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;
        struct vm_area_struct * vm_next;

        // Lista cykliczna (dla obszarów z i-węzłem i współdzielonych)
        struct vm_area_struct * vm_next_share;
        struct vm_area_struct * vm_prev_share;
        // tablica metod
        struct vm_operations_struct * vm_ops;
        // przesunięcie w pliku, od którego rozpoczyna się obszar
        unsigned long vm_offset;
        // i-węzeł pliku
        struct inode * vm_inode;
        // katalog stron
        unsigned long vm_pte;
};
```

Flagi obszaru: VM_READ, VM_WRITE, VM_EXEC, VM_SHARED (obszar reprezentuje odwzorowanie współdzielone) VM_GROWSDOWN - automatycznie rośnie w dół (stos i386) VM_LOCKED – stron nie można usunąć z pamięci (wymaga przywilejów superużytkownika).

Obszary pamięci c.d.

- Obszary możemy podzielić na dwie grupy.
  - Obszary związane z odwzorowaniem pliku w pamięci (a) utworzonym jawnie poprzez mmap, (b) utworzonym poprzez uruchomienie pliku wykonywalnego (c) utworzonym poprzez załadowanie biblioteki współdzielonej. W przypadku tego obszaru obsługa błędu braku strony może się wiązać z jej sprowadzeniem z pliku.
  - Obszary anonimowe, w których pole vm_inode==NULL. Nie są związane z odwzorowaniem pliku, brak strony oznacza konieczność sprowadzenie jej z pamięci wymiany.
- Zastosowanie drzew AVL. Kluczem w drzewie jest początek obszaru (vm_start). Podczas obsługi wyjątku stronicowania musimy odnaleźć obszar, którego dotyczy ten wyjątek. Robi to funkcja jądra

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

Zazwyczaj proces ma niewiele obszarów. Niektóre (np. obiektowe bazy danych) mogą jednak mieć ich kilka tysięcy. W takim przypadku wyszukiwanie w drzewie O(logN) jest znacznie szybsze niż wyszukiwanie liniowe O(N).

Obszary przykładowego procesu

- Program mc. Zawartość pliku /proc/59/maps

zakres adresów prawa offset urządzenie i-węzeł (0-obszar anonimowy)

```
08048000-080e9000 r-xp 00000000 03:01 35606 (kod programu)
080e9000-080f0000 rw-p 000a0000 03:01 35606  (dane)
080f0000-08110000 rwxp 00000000 00:00 0      (bss+sterta )
40000000-40005000 r-xp 00000000 03:01 43212
40005000-40006000 rw-p 00004000 03:01 43212
40006000-40007000 rw-p 00000000 00:00 0
40007000-40008000 r--s 00000000 03:01 59322
40008000-40081000 r-xp 00000000 03:01 7878  (biblioteka C - kod)
40081000-40087000 rw-p 00078000 03:01 7878  (biblioteka C - dane)
40087000-400ba000 rw-p 00000000 00:00 0     (biblioteka C - bss)
bfffd000-c0000000 rwxp ffffe000 00:00 0     (stos + środowisko)
```

- Prawa dostępu (rwx) + (p) – MAP_PRIVATE albo (s) MAP_SHARED
  - W procesorach Intela r implikuje x.
- offset przesunięcie od początku pliku (musi być wielokrotnością rozmiaru strony).

Metody obszaru

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

- Najważniejsze metody to:
  - nopage wywoływana przy braku strony.
  - wppage przy zapisie do strony chronionej.
  - swapin, swapout przy wymianie pamięć <=> wymiany
  - open, close – zliczanie referencji modułu w przypadku pliku specjalnego odwzorowanego w pamięć (odwzorowanie pozostaje w pamięci po zamknięciu pliku – szczegóły na następnym wykładzie)