Wykład 10
Zarządzanie pamięcią
część pierwsza: pamięć jądra

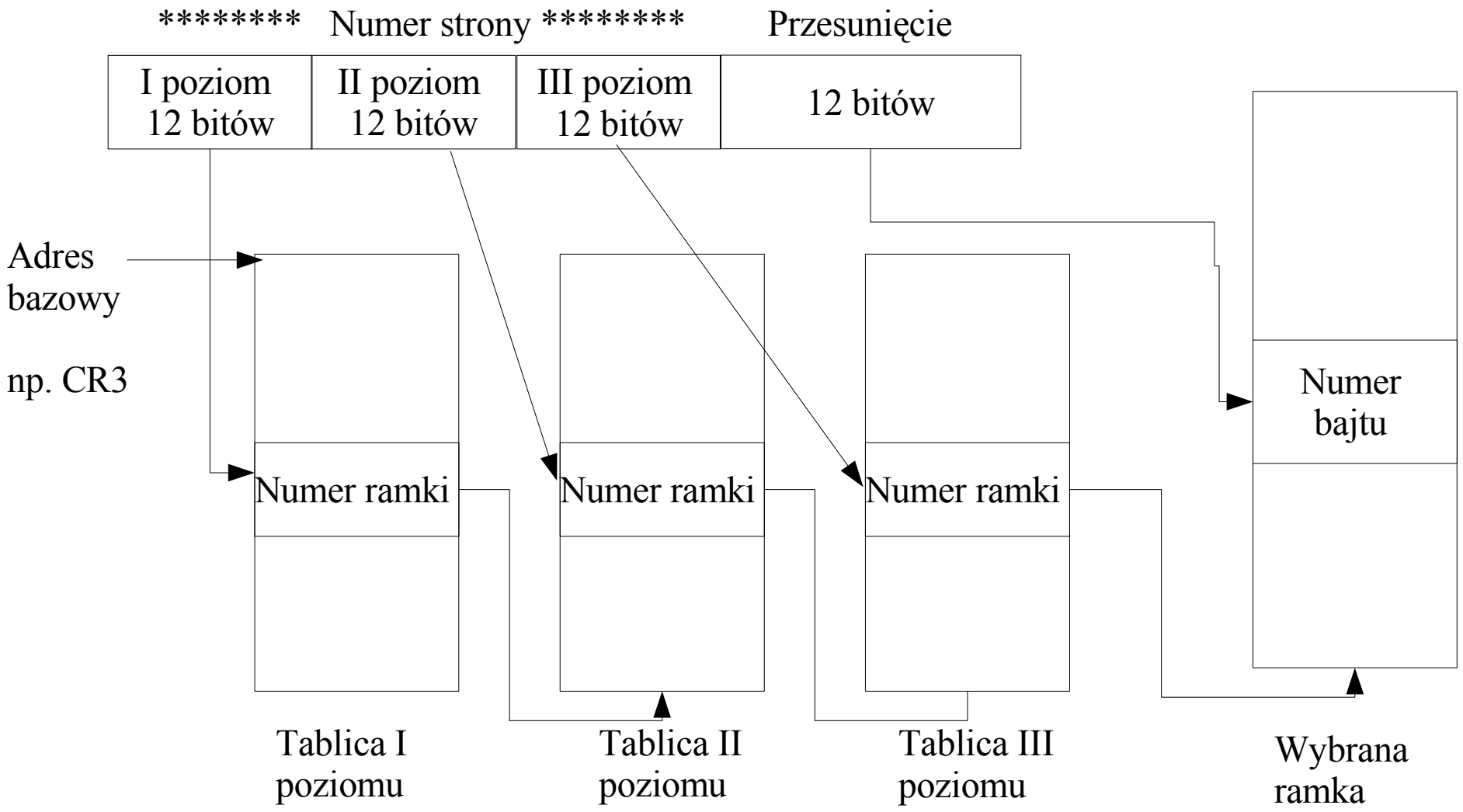
Sprzeczności przy projektowaniu systemu zarządzania pamięcią

- Implementuj możliwie wydajnie mechanizm pamięci wirtualnej.
- Przeznacz możliwie najwięcej pamięci RAM na
  - Pamięć wykorzystaną procesów.
  - Podręczną pamięć buforową.
  - Podręczną pamięć stron (jeszcze jej nie omawialiśmy)
- Problemem jest podział pamięci, pomiędzy te systemy tak, aby nowe żądania mogły być szybko zaspokojone.
  - Np. dla zwiększenia wydajności przydzieliliśmy prawie cały wolny RAM (Linuks to potrafi) na bufory.
  - Nagle natychmiast potrzebujemy nowej pamięci
  - Ale odnalezienie wolnej pamięci może potrwać.

Stronicowanie w Linuksie

- ***Uwaga: Nie ma stronicowania => Linux nie działa !!!***
- Przestrzeń adresów dzielona jest na strony (ang. page). Pamięć operacyjna na ramki (ang. frame).
  - Rozmiar strony == rozmiar ramki (np. 2,4,8 KB, u Intela 4KB lub 4MB)
- Adres wirtualny ma postać <numer strony, przesunięcie>, gdzie przesunięcie jest numerem bajtu na stronie.
  - Podczas translacji adresów numer strony jest zastąpiony przez numer ramki.
  - Dowolnej stronie może odpowiadać dowolna ramka.
- Najprostsze rozwiązanie: W pamięci istnieje tablica stron pozwalająca zamienić numer strony na numer ramki.
  - Każdy proces ma swoją własną tablicę stron. Na początek bieżącej tablicy wskazuje rejestr procesora (u Intela CR3). Wartość tego rejestru musi być więc zmieniona przy przełączeniu procesu.
- Policzmy na palcach: Adres ma 32 bity, przesunięcie 12 bitów, stąd numer strony 20 bitów stąd mamy 1M stron. Pozycja w tablicy stron zajmuje 4bajty, stąd każdy proces wymaga 4MB RAMu na tablicę stron ??? - nawet jeżeli zajmuje 4KB ?
- Na szczęście jest rozwiązanie: tablice wielopoziomowe

Stronicowanie wielopoziomowe



- Pentium i poprzednicy wykorzystują 2 poziomy
- Procesory 64 bitowe: Alpha, MIPS, Itanium, (Pentium Pro i dalsze opcjonalnie) wykorzystują 3 poziomy.
- Linux wewnętrznie wykorzystuje model 3 poziomowy

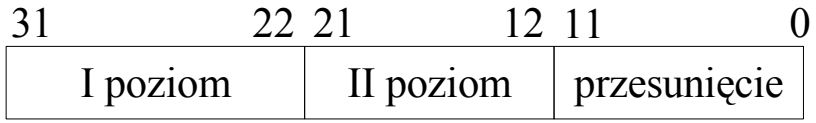
Bufor TLB.

- Policzmy raz jeszcze: aby odwołać się do pamięci muszę najpierw odwołać się do tablicy trzeciego poziomu, potem drugiego i na końcu trzeciego. Dopiero wtedy mogę odczytać dane.
  - Stronicowanie spowalnia pamięć czterokrotnie ???
- Na szczęście projektanci sprzętu rozwiązali ten problem przy pomocy bufora TLB (ang. Translation Lookaside Buffer). Jest to mała (np 64 pozycje) pamięć podręczna znajdująca się wewnątrz procesora przechowująca najczęściej wykorzystywane pary <numer strony, numer ramki>. Jeżeli numer strony jest w TLB (na ogół >90%), to procesor uzyskuje numer ramki z zerowym kosztem.
- Użycie TLB ma pewne (przykre) konsekwencje.
  - Zmiana w tablicy stron dotycząca strony która jest w TLB sprawia, że procesor może nie zauważyć tej zmiany i dalej korzystać z TLB => Należy unieważnić (ang. Invalidate) pozycję w tablicy.
  - Powyższy problem staje się koszmarny w systemach wieloprocesorowych, gdy jeden procesor zmienia tablicę stron drugiego procesora.
  - Po przełączeniu kontekstu zmieniamy całą tablicę stron trzeba opróżnić cały bufor TLB. Spowalnia to na pewien czas pracę systemu i zwiększa koszt przełączenia kontekstu.

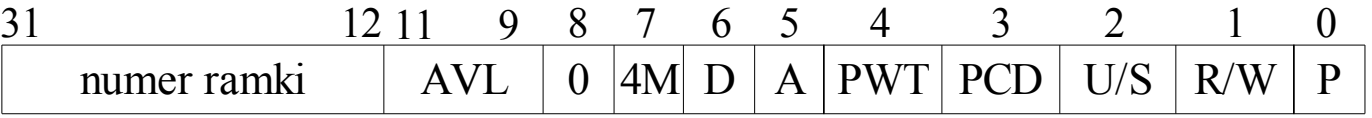
Pozycja tablicy stron w rzeczywistej maszynie

- Zawiera numer ramki.
- Zawiera tryb (jądra, systemu), uprawnienia (modyfikacja, wykonanie kodu) oraz bit obecności.
- Próba wykonania strony niezgodnie z ograniczeniami (tryb, uprawnienia) – powoduje wyjątek błędu strony.
- Próba wykonania strony z wyzerowanym bitem obeności również generuje błąd strony.
  - Jest to wykorzystane do ograniczenia rozmiaru pamięci procesu.
  - Jest to wykorzystane do implementacji pamięci wirtualnej.
  - Strony przechowywane na dysku mają wykasowany bit obecności. Handler obsługujący błąd strony przydziela wolną ramkę, ładuje stronę z dysku, ustawia bit ważności oraz numer ramkii powraca do programu użytkownika. Uwaga: W Linuksie na urządzeniu zewnętrznym mogą być przechowywane wyłącznie strony procesów użytkownika a nie jądra !!!
- Ponadto zawiera pola niezbędne do wydajnej implementacji pamięci wirtualnej.
  - Bit modyfikacji (ang. Dirty) jest ustawiany automatycznie w momencie zmiany zawartości strony. Służy do wyznaczenia stron, które wolno zwolnić dopiero po zapisie na dysk.
  - Bit dostępu (ang. Accessed) ustawiany automatycznie w momencie odwołania się do strony. Służy do wyboru strony ofiary, gdy w systemie brak pamięci, przez algorytm zastępowania stron (np. LRU i algorytmy aproksymujące np NFU).

Intel Pentium – stronicowanie dwupoziomowe



- Ramka ma ro zmiar 4KB (dwanaście bitów).
  - numer ramki 20 bitów
- Numer pierwszego i drugiego poziomu 10 bitów. Pozycja ma długość 4 bajtów.
  - Zatem tablica pierwszego i drugiego poziomu mieści się w jednej ramce !!!

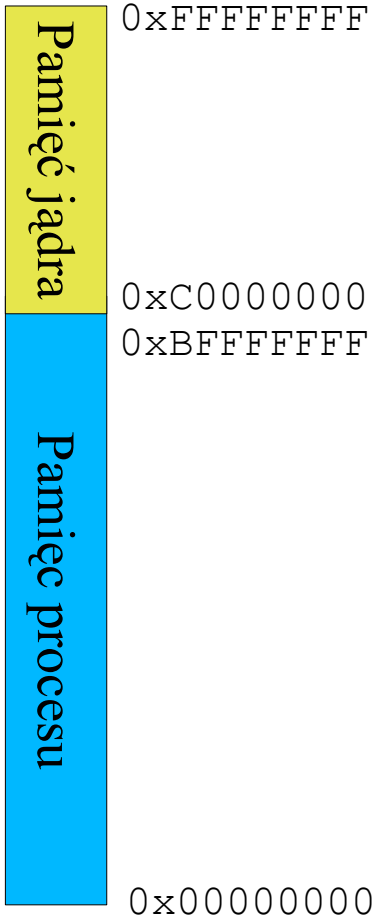


- AVL – available to OS, D – Dirty, A – available, PWT (page write through) PCD (page caching disable), U/S (user/supervisor) R/W (read /write) P – present.
- Nieobecność 4MB (1024 ramki) uzyskujemy kasując bit P w katalogu stron.
- Wartość efektywną bitów U/S oraz R/W jest sumą logiczną wartości pierwszego i drugiego poziomu.
- Bit 4M (tylko katalog stron, wprowadzony w Pentium) – oznacza pominięcie drugiego poziomu translacji. W tym przypadku przesunięcie ma 22 Bity, co oznacza stronę o długości 4MB. Jest on wykorzystywany dla dużych obszarów pamięci fizycznej w sposób ciągły odwzorowanych w przestrzeni adresowej (np. pamięć ekranu SVGA). Pozwala to na zoszczędzenie pozycji w buforze TLB

Abstrakcja tablic stron – przenośność pomiędzy maszynami

- plik ./include/asm-i386/page.h oraz ./include/asm-i386 pgtable.h
- type pgd_t reprezentuje element tablicy I poziomu, pmd_d element tablicy drugiego poziomu (middle – środkowy; nie obecny w x86), pte_t element tablicy stron.
- Makra i funkcje do manipulacji elementami tablic. Na przykład dla typu jeżeli x to pte_t mamy:
  - pte_dirty(x) sprawdza czy bit D jest ustawiony, pte_mkdirty(x) ustawia bit.
  - Podobnie pte_write, pte_read, pte_mkwrite, pte_mkread, pte_exec, pte_mkexec, pte_dirty, pte_referenced
  - pte_present, clear_pte
- Funkcja pte_t mk_pte(unsigned long page, pgprot_t pgprot) tworząca pozycje w tablicy na podstawie numeru ramki (page) oraz praw dostępu (pgprot)
- pgd_t * pgd_offset(struct mm_struct * mm, unsigned long address) na podstawie adresu w przestrzeni adresowej procesu (mm) znajduje pozycję głównego katalogu, której dotyczy ten adres.
- unsigned long pte_page(pte_t pte) – adres strony której dotyczy ta pozycja.

Podział przestrzeni adresowej procesu



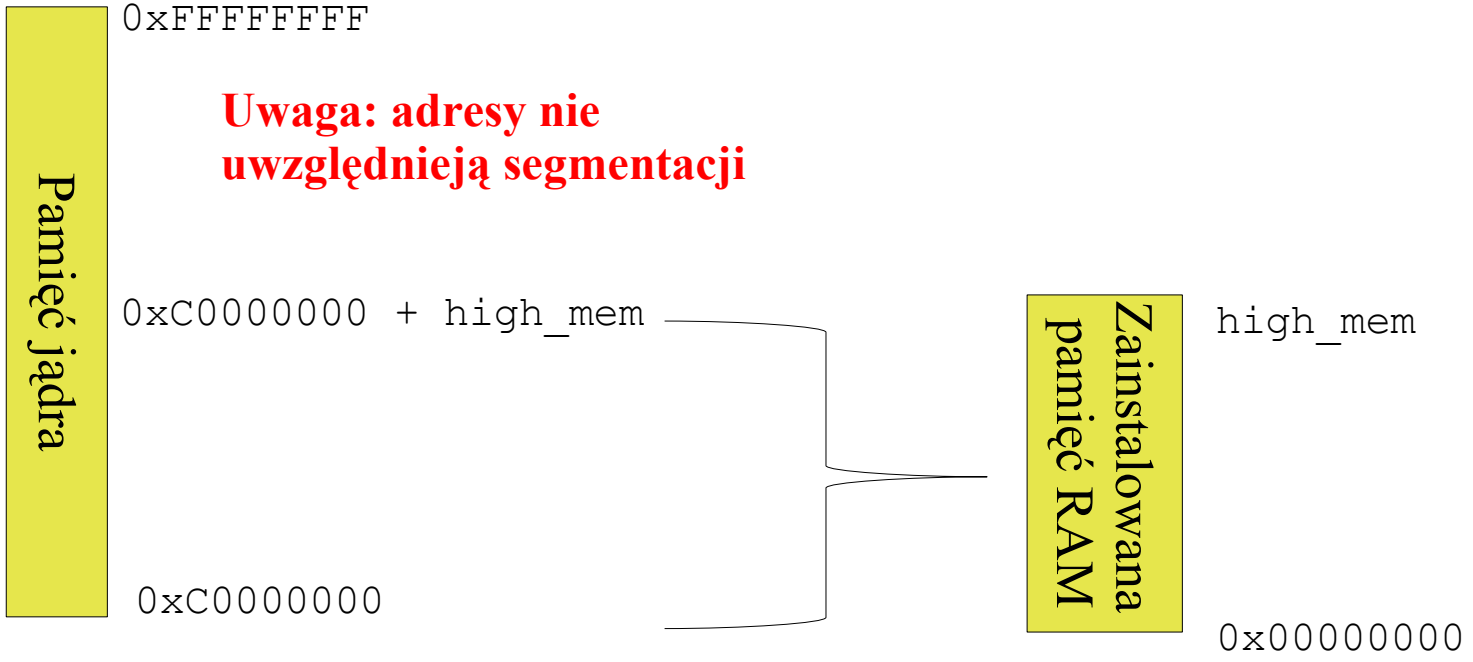
- Dolne 3GB (do adresu 0xBFFFFFFF) pamięć procesu: kod, dane oraz biblioteki współdzielone. Obszar ten dostępny jest z zarówno z trybu jądra jak i z trybu użytkownika.
- Górne 1GB. Kod i dane jądra. Obszar ten dostępny jest wyłącznie z trybu jądra !!!
- Każdy proces ma identyczne odwzorowanie górnego 1GB
  - Co oznacza, że maksymalnie 256 ostatnich pozycji w katalogu stron jest identyczne i wskazuje na tablice stron jądra (współdzielone przez wszystkie procesy).
  - Rejestr CR3 jest zmieniany z chwilą przełączenia kontekstu, nigdy z chwilą przejścia do jądra !!!
  - Jakiekolwiek zmiany odwzorowania górnego 1GB muszą być wykonane jednocześnie w tablicach stron wszystkich procesów (funkcje vmalloc, vremap, vfree).
- Na dzisiejszym wykładzie zajmiemy się górnym 1GB – pamięcią dla jądra

Wpływ segmentacji

- W architekturze x86 wykonywane są dwie translacje adresów: (a) segmentacja (b) stronicowanie.
- Pomijając szczegóły segmentacji w trybie chronionym (temat na odrębny wykład), można powiedzieć, że adres liniowy jest uzyskiwany poprzez dodanie do adresu logicznego początku segmentu. (jeżeli adres logiczny przekracza długość segmentu generowany jest wyjątek). Następnie adres liniowy jest zamieniany na adres fizyczny poprzez stronicowanie.
- W trybie użytkownika początek segmentu jest równy 0x00000000 (nic się nie zmienia)
- W trybie jądra następuje zmiana segmentu i początek segmentu jest równy 0xc0000000, a jego długość 1GB.
  - A zatem adres a zamieniany jest na adres C0000000+a.
- Segment użytkownika jest dostępny poprzez rejestr segmentu dodatkowego FS (stąd funkcje memcpy_from_fs).

Tablica stron pamięci jądra

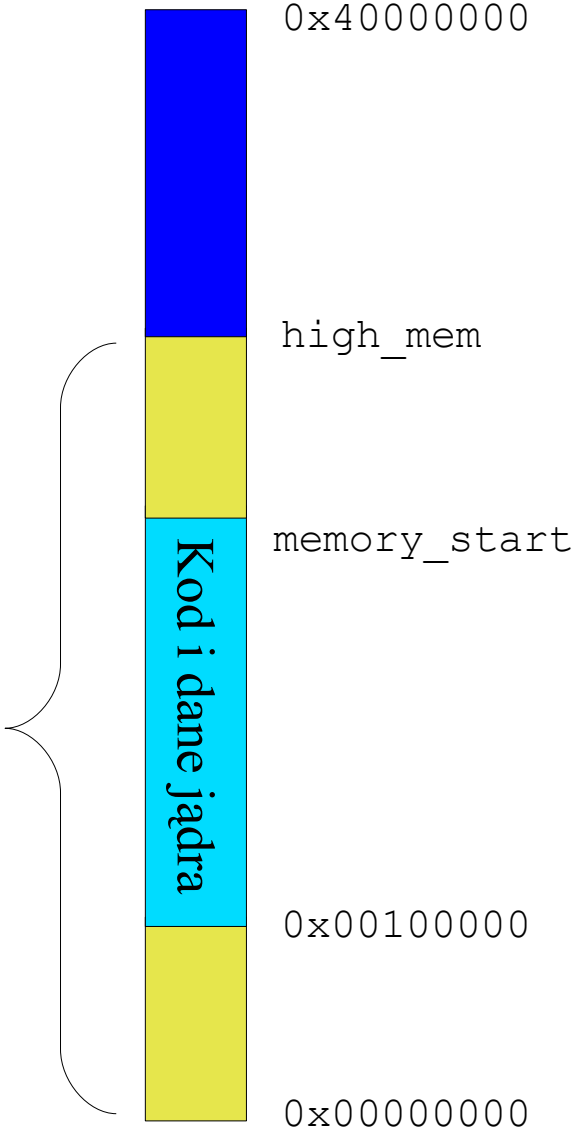
- Przy starcie systemu konstruowane jest następujące odwzorowanie pamięci jądra (górnego 1GB), gdzie high_mem jest rozmiarem RAMu w systemie -1.



Podsumowanie

- W trybie jądra (a) segmentacja zwiększa każdy adres o 0xC0000000 (b) stronicowanie zmniejsza każdy adres mniejszy od high_mem o 0xC0000000.
  - Zatem wpływ ten się znosi !!!
- Zatem w zakresie od <0 – high_mem> adres logiczny jest równy adresowi fizycznemu ***(tylko w trybie jądra)*** !!!
  - Jądro może zaadresować dowolny bajt pamięci operacyjnej RAM stosując adres fizyczny !!!
- Kontrolery DMA "nie widzą" translacji adresów.
  - Ale w zakresie <0 – high_mem> każdy adres zwrócony przez funkcje alokującą pamięć możemy przekazać urządzeniu wykorzystującemu sprzętowy kontroler DMA
- W trybie jądra dopuszczalny zakres adresów to 0 – 0x4000000. (górny gigabajt).
- Do pamięci procesu możemy odwołać się wykorzystując segment FS (memcpy_tofs , memcpy_fromfs, put_user, get_user).
  - W tym przypadku przesunięcie o 3GB w górę jest wyłączone.

Podział 1GB przestrzeni jądra



- Dla uniknięcia problemów z “dziurą”, kod jądra rozpoczyna się od pierwszego megabajta.
- to odwzorowana tożsamościowo pamięć RAM która może być przydzielana i zwalniana dynamicznie (np. przy pomocy kmalloc).
  - Także pamięć dla procesów przydzielana jest dla tego obszary, choć oczywiście pod innymi adresami.
- strony w tym obszarze są oznakowane jako nieobecne. W tym obszarze możliwe jest:
  - odwzorowanie pamięci urządzenia np. pamięci obrazu karty SVGA (vremap)
  - odwzorowanie stron pamięci RAM, aby uzyskać duży ciągły obszar w pamięci wirtualnej (vmalloc).
- Uwaga: w pojedyncza ramkę pamięci może być odwzorowanych kilka stron.
  - Np. każda ramka przydzielona procesowi i odwzorowana w jego pamięć jest też odwzorowana tożsamościowo.

Reprezentacja ramki w systemie

```
typedef struct page {
   // Zarządzanie listami (np lista wolnych stron)
   struct page *next;
   struct page *prev;
   // i-węzeł i przesunięcie z którego odczytana jest strona
   struct inode *inode;
   unsigned long offset;
   struct page *next_hash;
   // Liczba użytkowników: strony mogą być współdzielone
   atomic_t count;
   unsigned flags;
   unsigned dirty:16,age:8; // Używane przy wymianie stron
   struct wait_queue *wait; // Kolejka procesów oczekujących
   struct page *prev_hash;
   struct buffer_head * buffers; // Pierwszy bufor dyskowy na stronie
   unsigned long swap_unlock_entry;
   unsigned long map_nr;    // numer ramki
} mem_map_t;
```

- struct page = mem_map_t
- tablica mem_map, o rozmiarze równym liczbie ramek pamięci fizycznej.
- Makro MAP_NR(x) zwracające, dla adresu fizycznego x, indeks ramki opisującej ten adres.

Flagi ramki

- Kombinacja (suma logiczna) następujących stałych
- PG_locked – strona zablokowana w pamięci na czas trwania operacji wejścia-wyjścia.
- PG_referenced – wykorzystany przy algorytmie zastępowania stron (pobranie strony z pamięci podręcznej stron ustawia ten bit na jeden).
- PG_uptodate – strona zawiera poprawne dane w porównaniu z kopią w obszarze wymiany
- PG_DMA – strona z dolnych 16MB RAMu, dostępna do transmisji przez DMA na szynie ISA.
- PG_reserved – strona zarezerwowana, nie podlega dynamicznie alokacji, np. kod i dane statyczne jądra, “dziura” w zakresie 640KB-1MB.
- PG_free_after, PG_decr_after, PG_swap_unlock_after – związane z synchronizacją procesu przesyłania stron z i do pamięci pomocniczej.

Zarządzanie wolnymi stronami

- Wolne strony są przechowywane w blokach o rozmiarach 1, 2, 4, 8, 16, 32 stron. 32 strony=128KB.
- Listę wolnych bloków o danym rozmiarze przechowuje struktura

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

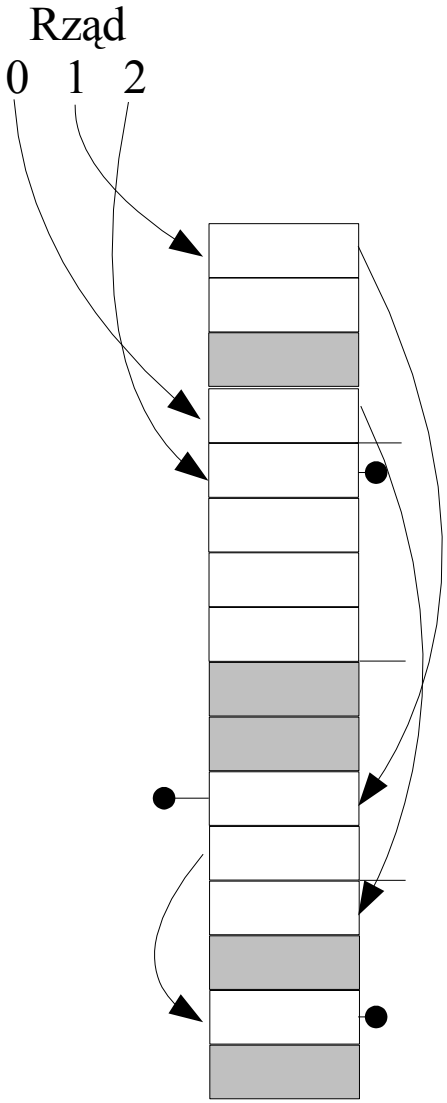
gdzie next jest wskaźnikiem do strony pierwszej strony pierwszego wolnego bloku. Pole next w tej stronie wskazuje na pierwszą stronę drugiego wolnego bloku. Map jest wskaźnikiem do mapy bitowej reprezentującej pamięć podzieloną na bloki danego rozmiaru. Tablica

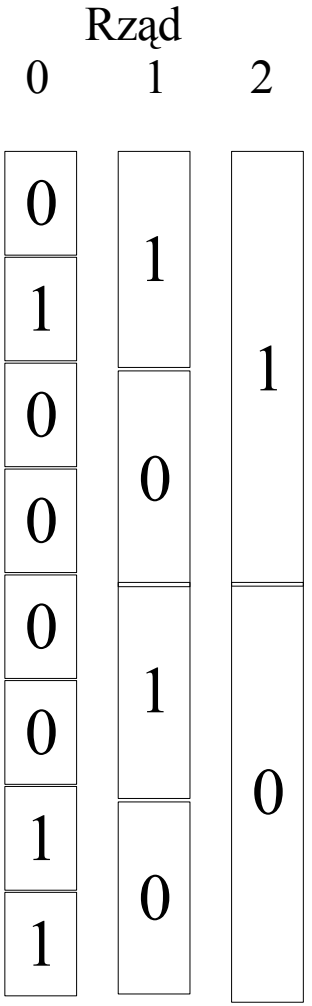
```
static struct free_area_struct free_area[6]
```

zawiera nagłówki sześciu list odpowiadającym 6 rozmiarom bloków (free_area[0] – 1 strona, free_area[1]-2, free_areas[2]-4).

- Postać map bitowych jest “ciekawa”

Mapy bitowe





- Kolor szary == strona zajęta
- Rząd 0 – jedna strona, 1 – dwie strony, 2 cztery strony. (w jądrze maks. 5 – 32 strony)
- Każda pozycja mapy odpowiada dwóm sąsiednim blokom.
- 1 oznacza, że jeden z tych bloków jest wolny.
- 0 obydwa (częściowo) zajęte.
- Dwa kolejne bloki nie mogą być wolne, bo zostałyby połączone w blok wyższego rzędu.
- Dzięki listom, odnalezienie wolnego bloku jest bardzo szybkie.

Przydział ramek

- Załóżmy, że potrzebujemy cztery stron. Przypuśćmy że najmniejszy wolny blok ma 16 stron. Dzielimy go na dwie części i:

Do listy wolnych ośmiostronicowych

- Dzielimy lewą połowę na dwie części (po cztery strony).

Zaalokowane cztery strony Do listy wolnych czterostronicowych

- Generalnie kontynuujemy podział tak długo, jak mamy do czynienia z blokami częściowo wolnymi i częściowo zajętymi.
- Koszt stały !!! (maksymalnie pięć kroków).

Strategia bliźniaków (ang .buddy)

- Opisany algorytm nosi nazwę strategi bliźniaków (ang. buddy, nie Buddy !!!).
  - Bliźniaków bo blok jest zawsze dzielony na dwa bliźniaki o równych rozmiarach.
- Przy zwalnianiu bloku stron sprawdzana jest możliwość łączenia bloków w większe.
  - np. po zwolnieniu przydzielonych w poprzednim przykładzie czterech stron otrzymalibyśmy z powrotem jeden blok 16 stronicowy.
  - Algorytm łączy bloki z sąsiadem tak długo jak to jest możliwe.
  - Również koszt stały
- Strategia bliźniaków stara się minimalizować fragmentację pamięci.
- Jednocześnie narzut czasowy związany z przetwarzaniem wolnych bloków jest minimalny.

Funkcje jądra do przydziału i zwalniania stron

```
// Pojedyncze strony
unsigned long __get_free_page(int priority);
void free_page(unsigned long addr);
// gwarantuje wyzerowanie strony
get_free_page(int priority)

// Bloki stron
unsigned long __get_free_pages(priority,order,dma);
void free_pages(unsigned long addr,int order);
```

- Order jest określa liczbę stron (potęga dwójki np. order=3 => 8 stron)
- dma =1 strony z dolnych 16MB RAM. Kontroler DMA na szynie ISA może adresować tylko taką pamięć.
- Prioryty:
  - GFP_KERNEL: może uśpić wołający proces (np. trzeba zapisać bufory dysku przed ich zwolnieniem)
  - GFP_ATOMIC: gwarantuje brak uśpienia procesu (można wywoływać wewnątrz handlera przerwania). Należy liczyć się z tym, że alokacja się nie uda.
- Jeżeli strona ma być dostępna z przestrzeni użytkownika, należy pamiętać o jej wyzerowaniu, chodzi o względy bezpieczeństwa !!!

Gdy potrzebujemy mniejszych bloków pamięci

- Posługujemy się funkcjami kmalloc/kfree.

```
// priority ma takie same znaczenie jak w __get_free_pages
void *kmalloc(size_t size, int priority)
void kfree(void *__ptr)
```

- Funkcje te wykorzystują bloki bajtów o rozmiarach (w przybliżeniu) równym kolejnym potęgom liczby dwa. Jednakże rozmiar bloku zmienia się od 32bajtów do 128KB.
  - Gwarantuje to fragmentację nie większą niż 50%.
- Przy alokacji dużych bloków pamięci, większych od romiaru strony, lepiej użyć __get_free_pages. (Np. kmalloc 64KB wykorzystuje 128KB RAM).
- Olbrzymią zaletą funkcji kmalloc/__getfree_pages jest fakt, że zarezerwowana pamięć obejmuje ciągły obszar pamięci fizycznej.
- Może być wykorzystana jako bufor urządzenia opartego na DMA.
- Jednakże długość przydzielanego bloku jest ograniczona do 128KB (kmalloc korzysta z __get_free_pages, które w naturalny sposób jest ograniczone największym dostępnym rozmiarem bloku).

Naprawdę olbrzymie bloki pamięci: alokacja przy starcie jądra

- Start jądra – plik ./init/setup.c, funkcja start_kernel wywołuje funkcje inicjujące poszczególne systemy.
- Każda z nich ma postać:
  - memory_start=foo_init(memory_start, memory_end);
  - memory_start to adres pierwszego wolnego bajtu, memory_end ostatniego
  - funkcja ma postać long foo_init(long start, long end);
- Jeżeli funkcja nie rezerwuję pamięci, powinna zwrócić wartość start.
- Jeżeli potrzebuje pamięci, w takim razie powinna zwrócić wartość memory_start + rozmiar potrzebnej pamięci.
- Jest to **jedyny sposób** na przydzielenie ciągłego bloku pamięci fizycznej o rozmiarze większym od 32 stron (128 KB). Niestety nie jest dostępny modułom.
- Pod koniec wywoływana jest funkcja mem_init, nadająca wszystkim ramkom od adresu 0x00100000 (1MB) do memory_start stan PG_Reserved, co oznacza że są pominięte przy mechanizmie dynamicznej alokacji.
  - Wywołanie własnej funkcji musi mieć miejsce przed wywołaniem mem_init

Funkcje vmalloc/vfree

```
void * vmalloc(unsigned long size)
void vfree(void * addr)
```

- Przydzielają i zwalniają obszar o rozmiarze ograniczonym rozmiarem dostępnej pamięci operacyjnej.
- Mogą uśpić proces
- Obszar jest ciągły wyłącznie w przestrzeni adresów jądra, a nie w pamięci fizycznej – nie można zastosować jako bufor urządzenia opartego na DMA.
  - Wykorzystanie wyłącznie do buforów zarządzanych programowo (np. nasz sterownik ring).
  - Wymaga modyfikacji katalogu stron ***wszystkich procesów !!!***
- Przy alokacjach bloków małych rozmiarów należy użyć kmalloc/kfree.
  - Ponieważ vmalloc zaokrągla rozmiar bloku do wielokrotności rozmiaru strony !!!
- Uwaga: jądro używa vmalloc do alokacji pamięci modułom. Nie wolno zatem w kodzie modułu deklarować sprzętowo dostępnych buforów w postaci zmiennych statycznych.
  - np zmienna globalna: `char bufor[1024];`
  - może być położona w dwóch nieciągłych ramkach pamięci fizycznej !!!

Funkcja vremap – dostęp do pamięci urządzeń

- Niektóre urządzenia zewnętrzne posiadają pamięć adresowaną przez procesor. Typowym przykładem są karty graficzne PCI posiadające pamięć ramki (ang. frame buffer).
- Jeżeli w jądrze chcemy odwołać się do tej pamięci, to musimy zmodyfikować tablicę stron jądra tak, ponieważ po inicjalizacji dostępna jest tylko pamięć operacyjna komputera.
- Dokonujemy tego przy pomocy funkcji:

```
void * vremap(unsigned long offset, unsigned long size)
```

gdzie offset jest początkiem bufora (adres fizyczny), a size jego rozmiarem

- Podobnie jak w przypadku vmalloc nie ma własności adres fizyczny == adres logiczny.
  - Ale fizyczny już znamy.
  - I podobnie do vmalloc modyfikuje katalogi stron wszystkich procesów.
- Odwzorowanie wykonane przy pomocy vremap usuwamy przy pomocy vfree.