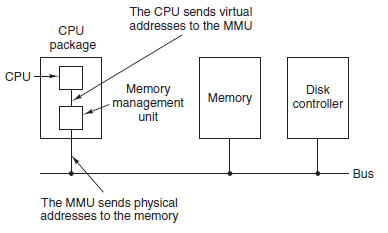
**Pamięć wirtualna**

**Pamięć wirtualna**

Idea: każdy proces otrzymuje swoją wirtualną przestrzeń adresową, posługują się adresami wirtualnymi. Procesor przy uzyciu MMU przy każdym dostępie do pamięci wykonuje translację adresów z wirtualnych na fizyczne.

Wirtualna przestrzeń adresowa jest podzielona na strony. Każda strona zawiera ciągły zakres adresów wirtualnych. Strony są mapowane na pamięć fizyczną, ale nie każda strona w wirtualnej musi znajdować się w fizycznej.

W komputerach bez pamięci wirtualnej adresy wirtualne są umieszczane bezpośrednio na magistralę pamięci, co powoduje odczyt lub zapis słowa pamięci wirtualnej o tym samym adresie.

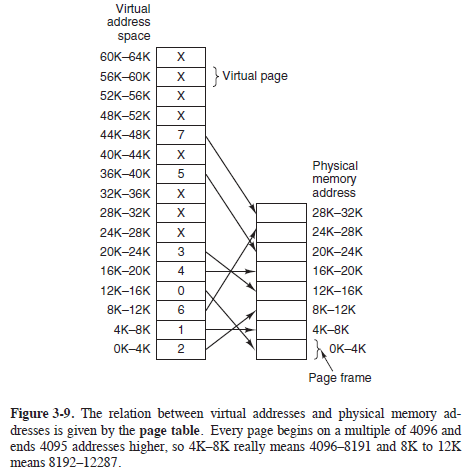
W przypadku zastosowania pamięci wirtualnej adresy wirtualne nie są przesyłane bezpośrednio na magistralę pamięci. Zamiast tego są one przesyłane do jednostki zarządzania pamięcią MMU, która odwzorowuje adresy wirtualne na adresy pamięci fizycznej. [](https://camo.githubusercontent.com/2496b31720d879fccaacd43f49c37c4c1d858e19/687474703a2f2f6936372e74696e797069632e636f6d2f326e39676865642e706e67)

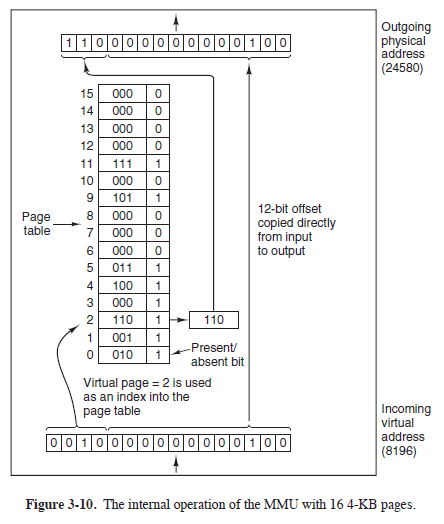
**Stronicowanie**

**stronicowanie** - technika zarządzania pamięcią, polegająca na podzieleniu na kawałki równej wielkości pamięci logicznej procesów (strony) i pamięci fizycznej (ramki); strony pamięci logicznej mogą być w dowolny sposób rozmieszczone w ramkach pamięci fizycznej.

W stronicowaniu cała przestrzeń adresowa jest podzielona na strony. Są to bloki pamięci ustalonego rozmiaru 2^k, które zaczynają się zawsze pod adresem podzielnym przez 2^k. Strony leżą w pamięci wirtualnej, a ramki stron w pamięci fizycznej.

**Translacja adresów**

Przykład odwzorowania adresów wirtualnych na adresy pamięci fizycznej: [](https://camo.githubusercontent.com/5f162a0ac940600e8f5366bbbd4f00277303cdf2/687474703a2f2f6936372e74696e797069632e636f6d2f323569367769782e706e67)

Zasada działania MMU: [](https://camo.githubusercontent.com/9b0e76a83d3ad617d05533189236bf2126f96ba1/687474703a2f2f6936332e74696e797069632e636f6d2f32647471396c642e706e67) Do jednostki MMU przesyłany jest wirtualny adres: 8196.

* 16-bitowy adres wirtualny jest dzielony na 4-bitowy numer strony i 12-bitowe przesunięcie
* numer strony jest wykorzystywany jako indeks do tabeli stron, która zwraca numer ramki strony odpowiadającej tej stronie wyrtualnej
* bit obecna/nieobecna:
  + 0 - wykonanie rozkazu pułapki do sysopka - brak strony (ang. *page fault*). Sysopek wybiera mało używaną ramkę strony i zapisuje jej zawartość na dysk (jeśli nie została zapisana tam wcześniej). Następnie pobiera stronę, do której przed chwilą program się odwoływał, na miejsce strony przed chwilą zwolnionej, modyfikuje mapę i wznawia przerwaną instrukcję.
  + 1 - to numer ramki strony znaleziony w tabeli stron jest kopiowany do 3 górnych bitów rejestru wyjściowego. Natomiast 12-bitowe przesunięcie jest kopiowane z wchodzącego adresu wirtualnego w postaci niezmodyfikowanej. Razem tworzą one 15-bitowy adres fizyczny.
* rejestr wyjściowy jest umieszczany na magistrali pamięci jako adres pamięci fizycznej

Tablice stron mogą być różnie zorganizowane: wielopoziomowa (Intel, ARM), odwrócona (PowerPC), wirtualna płaska (MIPS).

**Struktura wpisu w tablicy stron**

Wpisy tablicy stron oprócz fizycznego numeru strony zawierają:

* bit uprawnień - informuje, jakiego rodzaju dostęp jest dozwolony - czy procesor może odczytywać, zapisywać, wykonywać zawartość strony
* bit obecności - czy strona ma przypisaną ramkę
* bit trybu pracy procesora - czy procesor będąc w trybie usera albo nadzorcy może korzystać ze strony
* bity polityki buforowania - czy i jak procesor może przechowywać zawartość strony w pamięci podręcznej
* bity monitorowania dostępu
  + referenced - w użyciu - ustawiany, gdy następuje odczyt lub zapis strony. Wartość bitu pomaga w podjęciu decyzji o tym, którą stronę przerzucić na dysk w przypadku wystąpienia błędu braku strony- lepiej te, które nie są używane
  + modified - czy modyfikował tę stronę, zwany bitem zabrudzenia; bit ma sens, gdy sysopek zdecyduje się na odzyskanie ramki strony - kiedy strrna jest zapisywana, sprzęt automatycznie ustawia bit modified. Jeśli nie była modyfikowana (czyli jest "czysta"), to może zostać porzucona, ponieważ na dysku znajduje się prawidłowa kopia.

**TLB**

Obserwacja: większośc programów zazwyczaj wykonuje wiele odwołań do małej liczby stron. Tylko niewielka część pozycji w tabeli stron jest często czytana.

**TLB (*Translation Lookaside Buffer*)** - bufor, pamięć asocjacyjna, zwykle zlokalizowany w MMU, składa się z niewielkiej liczby pozycji. Każda pozycja to jedna strona - referenced, VPN, modified, uprawnień, PPN.

Zasada działania TLB:

* po przesłaniu adresu wirtualnego do MMU, aby dokonać translacji, najpierw sprawdzamy, czy w TLB jest VPN
  + TLB hit - ramka strony pobierana bezpośrednio z TLB
  + TLB miss - wyszukiwanie w tabeli, usuwa się jedną z pozycji w TLB i zastępuje odczytaną przed chwilą pozycją z tabeli stron

**zasięg TLB** - to rozmiar pamięci, którą możemy zaadresować bez generowania chybienia.

**miękkie chybienie** - nie ma w TLB, ale jest w pamięci, nie trzeba robić IO, tylko TLB update

**twarde chybienie** - nie ma w TLB, nie ma w pamięci, trzeba wziąć z dysku

**Błędy strony**

Gdy w wyniku przeszukiwania tablicy stron nie znaleziono strony w tablicy stron procesu, dochodzi do powstania błędu.

**drobny błąd strony (ang. *minor page fault*)** - strona może fizycznie być przechowywana w pamięci, ale nie ma jej w tablicy stron procesu. Strona mogła być np. załadowana do pamięci z dysku przez inny proces. W tym przypadku nie ma potrzeby, by sięgać do dysku ponownie. Wystarczy odpowiednio zmapować stronę w tablicy stron.

**poważny błąd strony (ang. *major page fault*)** - gdy zachodzi konieczność sprowadzenia strony z dysku

**błąd segmentacji (ang. *segmentation fault*)** - w programie nastąpiła próba dostępu do nieprawidłowego adresu i w ogóle nie ma potrzeby dodawania mapowania do TLB - sysopek zabija taki proces

**Zarządzanie cache'm a pamięcią wirtualną**

Realizujemy te same cele, ale między innymi poziomami hierarchii pamięci, tylko że tym razem robimy to programowo, a nie sprzętowo:

blok → strona metadane bloku → wpis tablicy stron pamięć asocjacyjna → tablica stron zastępowanie bloków → polityka zastępowania stron cache miss → major page fault pojemność pamięci podręcznej → rozmiar pamięci RAM pobranie bloku z RAM → pobranie strony z dysku polityka write-back → buforowanie stron

**Algorytmy zastępowania stron**

Chociaż istnieje możliwość wybrania losowej strony do usunięciaz pamięci przy każdym błędzie braku strony, wydajność systemu będzie znacznie lepsza, jeśli zostanie wybrana strona, która nie jest zbyt często wykorzystywana. W przypadku usunięcia często wykorzystywanej strony istnieje prawdopodobieństwo, że okaże się ona potrzebna w niedalekiej przyszłości.

**OPT**

Najlepszy możliwy algorytm jest łatwy do zapisania, ale niemożliwy do zaimplementowania.

Kiedy zachodzi page fault, w pamięci znajduje się pewien zbiór stron. Do jednej z tych stron nastąpi odwołanie w następnej instrukcji. Odwołanie do innych stron może nastąpić za 10, 100 lub 1000 instrukcji później. Każdą stronę można oznaczyć etykietą w postaci liczby instrukcji, które będą wykonane, zanim nastąpi pierwsze odwołanie do tej strony.

Z pamięci powinna być usunięta strona o najwyższej wartości etykiety. Jeśli jedna strona nie będzie używana przez następne 8kk instrukcji, a inne nie będzie używana przez 6kk instrukcji, usuwnięcie tej pierwszej odsuwa możliwość wystąpienia braku strony tak daleko w przyszłość, jak to możliwe.

Problem: system nie ma możliwości uzyskania informacji o tym, kiedy nastąpi kolejne odwołanie do każdej ze stron.

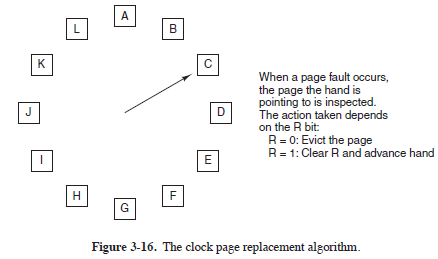
**FIFO**

Sysopek utrzymuje listę wszystkich stron, które aktualnie znajdują się w pamięci, przy czym strony dodane jako ostatnie znajdują się na końcu, natomiast te dodane najwcześniej - na początku. W przypadku page fault strona z początku listy jest usuwana, nowa dodawana na koniec.

**LRU**

Kiedy wystąpi błąd strony, wyrzucana jest strona, która nie była używana najdłużej.

**CLOCK**

FIFO + algorytm drugiej szansy + nieaktualizowanie listy stron [](https://camo.githubusercontent.com/905f416a3ad6ac7d136877ee24c2ba7b2cd2c8fb/687474703a2f2f6936342e74696e797069632e636f6d2f78346f7171772e706e67)

**L8.Z2.**

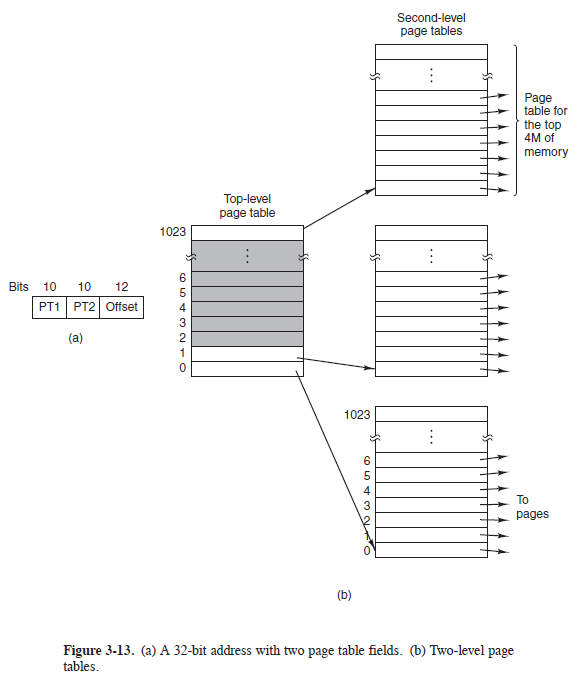
**Jak wspomniano na wykładzie wielopoziomowa tablica stron nie jest jedynym sposobem opisu wirtualnej przestrzeni adresowej. Przedstaw mechanizm odwróconej tablicy stron (ang. inverted page table). Wytłumacz jak przebiega proces tłumaczenia adresu wirtualnego na adres fizyczny. Jakie są wady i zalety takiej organizacji wpisów stron?** -> Podpowiedź: Porównanie najlepiej zrobić na podstawie §31.1.5 książki „Memory Systems: Cache, DRAM, Disk”

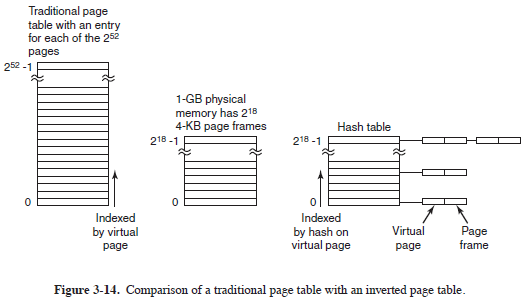
Póki przestrzenie adresowe w komputerach były małe, można było używać jednowymiarowych tablic stron. Przestrzenie zaczęły rosnąć -> tablice zaczęły rosnąć i zajmować dużo miejsca i trzeba było przenieść je do pamięci.

Dwa typy organizacji tablicy stron:

* forward-mapped lub hierarchical-mapped page table - indeksowana po VPN
* inverse-mapped lub inversed page table - indeksowana po PFN

**wielopoziomowa tablica stron** - Przy założeniu, że adresy są 32-b i strony 4-KB, to 4-GB przestrzeń adresowa jest złożona z 2^20 stron, tablica stron zajmuje 4 MB. To dużo, poza tym duża część stron nie będzie zmapowana (bo nie będzie używana - marnotrawstwo). Dlatego też stosuje się stronicowanie wielopoziomowe: chcąc sięgnąć do określonego słowa pamięci logicznej musimy zajrzeć kolejno do kilku mniejszych tablic stron. W przypadku architektury 32-bitowej są to dwa poziomy tablic stron. Załóżmy, że 32-bitowy adres logiczny jest podzielony na trzy części. Najstarsze bity określają pozycję w głównej tablicy stron (pierwszego rzędu). W tej tablicy znajduje się wskaźnik do tablicy stron drugiego rzędu. Środkowe bity adresu określają pozycję w tablicy stron drugiego rzędu, w której znajduje się już adres ramki. Najmłodsze bity określają adres w obrębie strony/ramki.

Na rysunku: 32-bitowy adres wirtualny podzielony na 10-bitowe pole TS1, 10-bitowe pole TS2 i 12-bitowe pole offset. [](https://camo.githubusercontent.com/c3c2f1b25dd404a34b31cdb423a3d62cdd256ebf/687474703a2f2f6936362e74696e797069632e636f6d2f317a70313772392e706e67)

**odwrócona tablica stron** - Jak jednak wygląda tablica stron w systemach w pełni 64-bitowych? 64 bity pozwalają zaadresować 16EB (czyli 17 179 869 184 GB). Bez względu na to, ile byśmy nie mieli poziomów stronicowania, tablica stron dla takiej przestrzeni adresowej (przyjmując wielkość strony 4kB) zajmowałaby ok. 32PB. Dlatego też, w systemach w pełni 64-bitowych stosuje się tzw. odwrotną tablicę stron. Odwrotna tablica stron zawiera tyle pozycji, ile ramek pamięci fizycznej jest faktycznie zainstalowanych w komputerze (lub wielokrotność tej liczby, np. czterokrotność). W każdej pozycji tablicy pamiętana jest para: numer strony i numer ramki, w której ona się znajduje. Dzięki temu, wielkość odwrotnej tablicy stron jest proporcjonalna do wielkości pamięci fizycznej faktycznie zainstalowanej w komputerze i stanowi jej niewielki promil. Jak jednak przetłumaczyć numer strony na numer ramki, nie przeglądając całej tablicy? Otóż odwrotna tablica stron jest tablicą mieszającą (ang. hash-table) -- strukturą słownikową o stałym oczekiwanym czasie dostępu. Dodatkowo, zwykle obliczanie funkcji mieszającej jest zaimplementowane sprzętowo w MMU, a odwrotnej tablicy stron towarzyszy szybka podręczna tablica stron. [](https://camo.githubusercontent.com/1cacdd528b7181b3a98a8771c73a83ccd18615ab/687474703a2f2f6936372e74696e797069632e636f6d2f317a64353636682e706e67)

**L8.Z3.**

**Dla podanego poniżej fragmentu wydruku polecenia «x86info -a» podaj zasięg bufora translacji adresów pierwszego i drugiego poziomu. Z jakich przyczyn producenci procesorów wprowadzili duże strony (ang. superpages, huge pages)? W jaki sposób zarządzać przydziałem dużych stron dla procesów? Jakie problemy są z tym związane?** L1 Data TLB: 4KiB pages, 16 entries L1 Data TLB: 4MiB pages, 4 entries  
L2 Data TLB: 4KiB pages, 256 entries L2 Data TLB: 4MiB pages, 32 entries

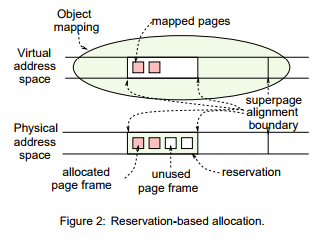
4KB \* 16 + 4MB \* 4 = 64KB + 16 MB = 16MB + epsilon 4KB \* 256 + 4MB \* 32 = 1MB + 128MB = 129MB -> Podpowiedź: Przeczytaj §2 publikacji “Practical, transparent operating system support for superpages”.

Motywacja: Zapotrzebowanie na RAM przez aplikacje rośnie, więc RAM w naszych komputerach rośnie. TLB nie zwiększa się. Jako tablica asocjacyjna, dostęp do niej musi być szybki, więc TLB nie może być duże. Pokrycie TLB maleje z każdym rokiem. W konsekwencji zbiór roboczy zaczyna przerastać TLB. Cache robią się większe, więc missy w TLB są kosztowne.

Metoda na zwiększenie pokrycia TLB (a nie rozmiaru TLB): chcemy używać stron o dużych rozmiarach ok, bo pokrycie TLB jest większe

Ograniczenia dla superstron:

* rozmiar dużej strony musi być rozmiarem obsługiwanym przez procesor
* duża strona musi być spójna i ciągła w fizycznej i wirtualnej przestrzeni adresowej
* w fizycznej i wirtualnej przestrzeni adresowej adres strony musi być podzielny przez jej rozmiar - 64KB superpage musi być wyrównany do 64KB address boundary
* wpis w TLB dla superstrony zawiera tylko pojedynczy referenced bit, modified (dirty) i bity zabezpieczeń. To oznacza, że wszystkie strony składające się na dużą stronę muszą mieć te same zabezpieczenia (read, write, execute). Jesteśmy w stanie stwierdzić, że jakaś część dużej strony była modifykowana, ale nie potrafimy powiedzieć, która to była dokładnie strona

Zarządzanie dużymi stronami można podzielić na serię kroków, każdy z nich wymaga pewnych kompromisów. Zakładamy, że przestrzeń wirtualna procesu składa się ze zbioru obiektów pamięci wirutalnej. Obiekt pamięci zajmuje ciągły obszar przestrzeni adresowej i wygląda tak: [](https://camo.githubusercontent.com/c84eaddf62a379a489351f99116be01f9c192d11/687474703a2f2f6936382e74696e797069632e636f6d2f7263376163352e6a7067)

Przykładami obiektów pamięci są kod, dane, stos, segmenty sterty procesów. Fizyczna pamięć dla tych obiektów jest alokowana, gdy pierwszy raz się do nich odwołujemy. Dla obiektów o stałym rozmiarze rezerwujemy tylko tyle ile trzeba, dla tych o rosnącym rozmiarze (sterta) alokujemy ile możemy, jak coś to preempt reservation.

Alokacja:

1. relocation-based Kiedy pierwszy raz program odwołuje się do obiektu pamięci, sysopek alokuje fizyczną ramkę pamięci. W zasadzie każda wolna ramka może być użyta w tym celu, tak jak w systemie bez wsparcia dla dużych ramek. Jednakże, może być konieczna relokacja ramek, w celu zachowania ciągłości i wyrównania dla dużych stron.
2. reservation-based Sysopek probuje zaalokwać ramkę, która jest częścią wolnego, ciągłego obszaru ramek o wielkości i wyrównaniu pasującym do dużej strony. Rezerwuje od razu cały zbiór na użytek procesu. Z góry trzeba wiedzieć, jaki rozmiar będzie miała duża strona, żeby tyle miejsca zarezerwować. Gdy będzie chciał już stworzyć dużą stronę, będzie miał to za darmo - pamięć jest zarezerwowana.

Kontrola fragmentacji: By uniknąć fragmentacji, sysopek może jednak zwolnić kawałek częściowo zajętego zarezerwowanego obszaru, co w konsekwencji może doprowadzić do tego, że duża strona nigdy nie powstanie.

Awans Jeśli sysopek zauważy, że jest ładny, ciągły, dobrze wyrównany zbiór ramek, może go upgradnąć do superpage. To powoduje update PTE dla każdej ramki, która wchodzi w skład dużej strony. TLB teraz w jednym wpisie ma całą dużą stronę. Sysopek może też stopniowo kolejne podzbiory awansować do minidużychstron, potem awansować je w większe aż do ostatecznej dużej strony. Trzeba wybrać, czy bardziej nam się teraz opłaca szybki awans - zredukowana liczba TLB miss, ale też większe pożeranie pamięci, jeśli nie wszystkie strony zawierające się w dużej stronie są wykorzystywane.

Degradacja Duże strony można degradować, albo odłączając poszczególne strony zawierające się w super page, albo dzieląc ją na mniejsze poddużestrony. Problem: sysopek utrzymuje tylko jeden referenced bit dla dużej strony, więc trudno mu sprawdzić, która część dużej strony jst aktywnie używana, a która nie.

Wykwaterowanie Nieaktywna duża strona może być wywalona z pamięci fizycznej, powodując, że wszystkie zawierające się w niej strony będą niedostępne. Jeśli będzie potrzebna ponownie, można dużą stronę stworzyć tak samo, jak podano wcześniej.

**L8.Z4.**

**Wydrukuj informacje dotyczące przestrzeni adresowej procesów użytkownika. Co opisują pola VmPeak i VmSize? Jak znaleźć rozmiar zbioru rezydentnego? Ile ramek przeznaczono na pamięć anonimową, a ile na odwzorowanie plików w pamięć? Jakiego rozmiaru są segmenty danych, kodu i stosu? Ile pamięci jądro zużywa na tablice stron opisujących przestrzeń adresową procesu?**

wszystkie procesy: ps -A wydruk: cat /proc/2204/status wydruk i dane info: cat /proc/\*/status konkretny proces i element: cat /proc/2204/status | grep VmPTE

**VmPeak/VmSize** - rozmiar zajmowanej pamięci wirtualnej peak/current total address space used (i.e sum of all virtual memory areas in the process)

**VmHWM/VmRSS** - rozmiar zbioru rezydentnego peak/current usage of physical RAM (HWM - High Water Mark) VmRSS - note that the value here is the sum of RssAnon, RssFile, and RssShmem.

**RssAnon** - ile ramek przeznaczono na pamięć anonimową size of resident anonymous memory

**RssFile** - ile ramek przeznaczono na odwzorowanie plików w pamięć size of resident file mappings

**VmData, VmStk, VmExe** - jakiego rozmiaru są segmenty danych, kodu i stosu size of data, stack and text segments

**VmPTE** - ile pamięci jądro zużywa na tablice stron opisujących przestrzeń adresową procesu Page Table Entries size

**L8.Z5.**

**Rozważmy system ze stałym rozmiarem zbioru rezydentnego. Mamy pamięć fizyczną składającą się z 4 ramek oraz pamięć wirtualną złożoną z 8 stron. Startujemy z pustą pamięcią fizyczną. Jedyny wykonujący się proces generuje następujący ciąg dostępów do stron: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Zaprezentuj działanie następujących algorytmów zastępowania stron: OPT, FIFO, LRU, CLOCK. Który z nich generuje najmniej błędów stron? Załóż, że zastąpienie ramki zachodzi dopiero w momencie zapełnienia całej pamięci fizycznej**

**zbiór roboczy** - to zbiór stron, których proces potrzebował w chwili t do działania w ciągu Δ ostatnich tyknięć wirtualnego zegara. Podzbiór rezydentnego w delcie czasu.

**zbiór rezydentny** - to zbiór wszystkich stron procesu rezydujących w pamięci operacyjnej w chwili t wirtualnego zegara procesu. System operacyjny zarządza zbiorem rezydentnym i stara się przybliżać nim zbiór roboczy. (Zbiór rezydentny to to co proces trzyma w ramie, wszystkie strony które sobie zadeklarował, jakby zamrozić całkowicie proces, i spojrzeć na cały RAM jaki on używał, to to jest zbiór rezydentny)

**szamotanie** - gdy proces ma mniej ramek niż liczba aktywnie używanych stron, dochodzi do szamotania. stan procesu, w którym spędza on więcej czasu na oczekiwaniu na brakujące strony pamięci niż na faktycznym wykonywaniu obliczeń, co znacząco spowalnia jego działanie.

OPT: 7: (-, -, -, 7) 0: (-, -, 0, 7) 1: (-, 1, 0, 7) 2: (2, 1, 0, 7) 0: hit 3: miss (((2, 2), (1, inf), (0, 1), (7, inf)) (2, 1, 0, 3) 0: hit 4: miss (((2, 1), (1, inf), (0, 3), (3, 2)) (3, 4, 0, 3) 2, 3, 0, 3, 2: hit

FIFO: 7: 0: 1: 2: (2, 1, 0, 7) 0: hit 3: miss (2, 1, 0, 3) 0: hit 4: miss (2, 1, 4, 3) 2, 3: hit 0: miss (2, 0, 4, 3) 3, 2: hit

LRU: 7: 0: 1: 2: (2, 1, 0, 7) 0: hit 3: miss (2, 1, 0, 3) 0: hit 4: miss (2, 4, 0, 3) 2, 3, 0, 3, 2: hit

CLOCK: 7: 0: 1: 2: ((7, 0), (0, 0), (1, 0), (2, 0)) 0: ((7, 0), (0, 1), (1, 0), (2, 0)) 3: miss ((0, 1), (1, 0), (2, 0), (3, 0)) 0: ((0, 2), (1, 0), (2, 0), (3, 0)) 4: miss ((0, 1), (4, 0), (2, 0), (3, 0)) 2: ((0, 1), (4, 0), (2, 1), (3, 0)) 3: ((0, 1), (4, 0), (2, 1), (3, 1)) 0: ((0, 2), (4, 0), (2, 1), (3, 0)) 3: ((0, 1), (4, 0), (2, 1), (3, 1)) 2: ((0, 1), (4, 0), (2, 2), (3, 0))

**L8.Z8.**

**W jaki sposób jądro Linuksa reaguje na wyczerpanie pamięci fizycznej? Na podstawie opisu Out-of-Memory Killer omów kryteria, którymi kieruje się jądro Linuksa w odzyskiwaniu pamięci fizycznej w krytycznych sytuacjach.**

Linuksowy "oom killer" jest odpowiedzialny za poświęcenie jednego lub więcej procesów w celu zwolnienia pamięci. To oczywiście zabije też każdy proces dzielący ten sam mm\_struct co wybrany zabijany proces. Proces może być uodporniony na działanie oom killera, jeśli wartość jego /proc/pid/oomadj jest ustawiona na stałą OOM\_DISABLE. OOM killer pozwala wybrnąć z szamotania - wybiera proces ofiarę i ją zabija.

Funkcja, która ocenia, czy proces jest dobrym kandydatem na bycie wyeliminowanym to badness(), która jest wynikiem następującego ciągu wywołań: \_alloc\_pages -> out\_of\_memory() -> select\_bad\_process() -> badness()

Komentarze do funkcji badness() mówią, co funkcja robi:

oom\_badness - oblicza, jak bardzo zadanie jest złe. Chcemy wybrać odpowiednie zadanie do zabicia. Odpowiednie znaczy:

* nie ma szkody takiej, że proces się dużo napracował, a teraz musi zostać zabity
* chcemy odzyskać dużą ilość pamięci
* nie zabijamy niczego, co nie zżera dużo pamięci
* chcemy zabić jak najmniej procesów (najlepiej jeden)
* zasada najmniejszej niespodzianki - user spodziewa się, że ten proces zostanie zabity

badness() sumuje "punkty" każdemu procesowi, który sprawdza i zwraca je do select\_bd\_process(). Proces z największą liczbą punktów jest eliminowany, chyba że właśnie sam grzecznie zwalnia pamięć.

Przydzielanie punktów

* naliczanie punktów zaczyna się od dodania rozmiaru pamięci zajmowanej przez proces
* potem rozmiar każdego dziecka - procesy forkujące duże dzieci są dobrym kandydatem
* krótko trwające procesy zazwyczaj są mało ważne, więc dublujemy ich punktacę
* procesy roota są zazwyczaj ważne, więc odejmujemy im punkty
* ostatecznie punktacja jest sziftnięta bitowo przez ustalaną przez usera stałą w /proc/pid/oomadj

Idealny kandydat do likwidacji niedawno się zaczął, nie jest procesem uprzywilejowanym, ma dużo dzieci i razem z nimi zadjmuje dużo pamięci.