8. ZARZĄDZANIE PAMIĘCIĄ

Do jednostki pamięci dociera tylko strumień adresów.

Nie dochodzą informacje o sposobie tworzenia adresów ani czego dotyczą.

Program rezyduje na dysku, jako plik binarny.
Po wprowadzeniu do PAO staje się procesem.
Proces w trakcie wykonywania może być przemieszczany między dyskiem a PAO.

Kolejka Wejściowa: zbiór programów (procesów) czekających na dysku na wykonanie.

Procesy mogą przebywać w dowolnej części PAO (pamięci fizycznej - RAM).

Program źródłowy przechodzi przez **kilka faz**, podczas których **ulega zmianie** jego lokalizacja.

Program źródłowy zawiera adresy wyrażone w sposób **symboliczny**.

Kompilator wiąże adresy symboliczne z adresami **względnymi**.

Konsolidator lub **program ładujący** wiąże adresy względne z **bezwzględnymi**.

Każde wiązanie to odwzorowanie z jednej przestrzeni adresowej na inną.

Kompilacja: Jeśli znane jest **miejsce** lokalizacji procesu w PAO, to można wygenerować **kod bezwzględny** *(absolute code).*

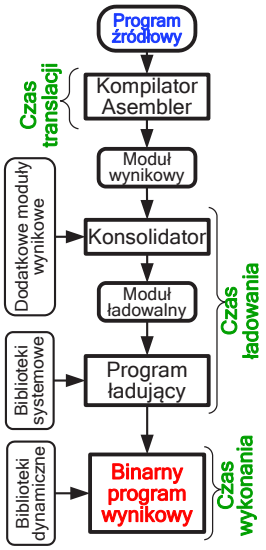
Gdy adres początkowy ulegnie zmianie, kod taki trzeba ponownie skompilować.

W MS-DOS pliki typu **.com** zawierają programy z adresami bezwzględnymi.

Ładowanie: Jeśli podczas kompilacji **nie wiadomo**, gdzie będzie umieszczony proces w pamięci, to kompilator wytwarza **kod przemieszczalny** *(relocatable code).*

Ponownego załadowania kodu wymaga tylko podania nowego adresu początkowego.

Wykonanie: Jeśli proces może **ulegać przemieszczeniom** w PAO podczas swojego wykonania, to trzeba czekać z wiązaniem adresów aż do czasu wykonania.



❒ Ładowanie dynamiczne

Tylko potrzebny podprogram zostanie wprowadzony do PAO.

Podprogramy na dysku przechowuje się w postaci przemieszczalnej.

Do pamięci wprowadza się wyłącznie program główny.

Chcąc wywołać inny podprogram należy sprawdzić, czy znajduje się on w PAO.

Jeśli nie, to trzeba wywoływać program łączący i ładujący moduły przemieszczalne, oraz uaktualnić tablicę adresów programu.

Programista decyduje czy jego program będzie mógł korzystać z ładowania dynamicznego.

SO dostarcza procedur bibliotecznych do realizacji ładowania dynamicznego.

❒ Konsolidacja dynamiczna

Biblioteki przyłączane dynamicznie *(dynamic linked libraries).*

Dotyczy to zwykle bibliotek systemowych, na przykład bibliotek języków programowania.

Jeśli system nie ma tej właściwości, to wszystkie programy muszą mieć dołączoną do swoich obrazów binarnych kopię biblioteki języka (lub kopie podprogramów, do których się odwołują).

Przy konsolidacji dynamicznej obraz binarny - w miejscu odwołania bibliotecznego - zawiera tylko ***zakładkę procedury,*** mały fragment kodu wskazująca jak odnaleźć podprogram biblioteczny.

☛ Zakładka wprowadza na swoje miejsce ***adres podprogramu*** i powoduje jego wykonanie.

Bibliotekę można zastąpić nową wersją i wszystkie odwołujące się do niej programy będą automatycznie używały nowej wersji.

Bez dynamicznej konsolidacji dostęp do nowych bibliotek, wymagałby ponownej konsolidacji programów.

Informację o ***wersji*** biblioteki dołącza się do programu i do biblioteki.

SO udziela pomocy przy realizacji konsolidacji dynamicznej.

➔ Procesy w pamięci chronione są przed sobą wzajemnie i tylko SO może zezwolić, aby wiele **Procesów** miało dostęp do tych samych adresów pamięci.

❒ Nakładki

Nakładki *(overlays)* umożliwiają wykonanie procesu większego niż dostępna pamięć.

W pamięci przechowuje się kod i dane, które są stale potrzebne.

Inne moduły wprowadzane są w miarę potrzeby na miejsca zajmowane przez już wykorzystane.

Rozważmy dwuprzebiegowy asembler.

Pierwszy przebieg generuje tablicę symboli,	kod przebiegu **I**	70 KB.
Drugi przebieg generuje kod maszynowy,	kod przebiegu **II**	80 KB.
Tablica symboli		**20 KB.**
Wspólne podprogramy dla obu przebiegów		**30 KB.**

Mając do dyspozycji **150 KB** nie można wykonać procesu asemblacji (potrzebuje 200 KB).

Kody przebiegu I i II nie muszą znajdować się w pamięci w tym samym czasie.

☛ Definiujemy dwie nakładki **A** (120 KB) i **B** (130 KB) oraz moduł obsługi nakładek (10 KB):

Nakładki **A** i **B** zawierają tablicę symboli, wspólne podprogramy i kod przebiegu **I** oraz **II**.

Po zakończeniu **przebiegu I** następuje skok do modułu obsługi nakładek, który na miejsce nakładki **A** czyta do pamięci nakładkę **B**, po czym rozpoczyna **przebieg II**.

Kody nakładek **A** i **B** przechowywane są na dysku w postaci obrazów bezwzględnych pamięci.

Nakładki nie wymagają specjalnego wsparcia ze strony systemu operacyjnego.

System operacyjny zauważa tylko zwiększoną liczbę operacji We/Wy.

Dlaczego przy **Nakładkowej** technice programowania SO rejestruje zwiększenie operacji WE/Wy ?

❒ Przestrzeń adresowa

Każdy proces otrzymuje własną wirtualną i **niezależną przestrzeń adresową**

Wątek wykonywany w danym procesie korzysta ***tylko*** z pamięci należącej do tego procesu.

Pamięć, która należy do pozostałych procesów, jest niewidoczna i niedostępna dla tego wątku.

Dla procesów 32-bitowych przestrzeń adresowa wynosi **4 GB**, gdyż 32-bitowy wskaźnik daje 4 294 967 296 różnych wartości.

Wirtualna przestrzeń adresowa (to **nie** fizyczna) to tylko **zakres adresów pamięci**.

> Aby z **wirtualnej pamięci** skorzystać (bez wywołania błędu dostępu), trzeba najpierw przypisać do fragmentu **pamięci wirtualnej** **pamięć fizyczną** czyli dokonać **mapowania**.

➔ **Program ma dostęp tylko do wirtualnej pamięci, natomiast *SO* mapuje adresy wirtualne w fizyczne** (nie muszą one stanowić ciągłego bloku).

Gdy proces ładowany jest po raz pierwszy, względne odwołania do pamięci w jego kodzie zastępowane są odwołaniami do adresów absolutnych w PAO, określonych przez **adres bazowy** ładowanego procesu.

Adres logiczny *(logical address)*: wytworzony przez CPU *(zwany adresem wirtualnym).*

Adres fizyczny *(physical address)*: umieszczony w *rejestrze adresowym* PAO.

- Adresy logiczne i fizyczne są **takie same** podczas **kompilacji** oraz **ładowania**.
- Adresy logiczne i fizyczne **są różne** podczas wykonywania rozkazów.

Logiczna przestrzeń adresowa: zbiór adresów logicznych generowanych przez program.

Fizyczna przestrzeń adresowa: zbiór adresów fizycznych odpowiadających adresom logicznym.

Proces **A** ma **strukturę danych** zapisaną w swojej przestrzeni adresowej pod adresem 0x12345678.

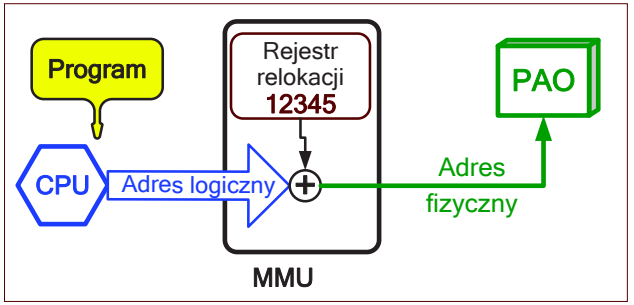
Proces **B** ma pod tym samym adresem, 0x12345678 zapisaną inną strukturę danych.

Wątki wykonywane w procesie **A**, sięgając do adresu pamięci 0x12345678, korzystają ze struktury danych procesu **A**.

Wątki wykonywane w procesie **B**, sięgając do adresu pamięci 0x12345678, korzystają ze struktury danych procesu **B**.

Jednostka zarządzania pamięcią (***M**emory **M**anagement **U**nit* - **MMU**)

Jest to urządzenie **sprzętowe** odwzorowujące adresy wirtualne na fizyczne w czasie działania programu.



Rejestr **bazowy** ***(relocation** register)* to **rejestr przemieszczenia**

Wartość **rejestru relokacji** dodaje się do każdego adresu wytwarzanego przez proces użytkownika, gdy odwołuje się do PAO.

Jeśli **baza** = **12345** i proces adresuje komórkę **55**, to adres ten jest dynamicznie zmieniany na odwołanie do komórki **12400 PAO**.

Program użytkownika działa na **adresach logicznych**, **nigdy** na ***rzeczywistych*** adresach fizycznych.

Sprzęt odwzorowujący pamięć zamienia adresy logiczne na adresy fizyczne.

Logiczna przestrzeń adresowa powiązana z odrębną, fizyczną przestrzenią adresową jest podstawą zarządzania pamięcią.

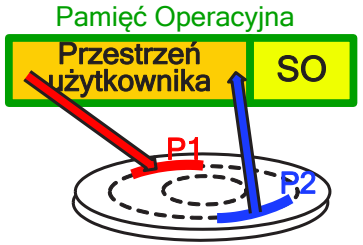
❒ Wymiana

Wykonanie procesu jest możliwe wtedy, gdy jest on w PAO.

Proces może być tymczasowo **wymieniany** *(swapped),* tj. odsyłany z PAO na dysk i z powrotem.

Po wyczerpaniu **kwantu czasu** zarządca pamięci rozpoczyna wymianę procesu.

W międzyczasie planista przydziału CPU przydzieli kwant czasu innemu procesowi w PAO.



Każdy proces po zużyciu kwantu czasu może zostać wymieniony z innym procesem.

Zarządca pamięci powinien wymieniać procesy szybko, aby w PAO zawsze były procesy gotowe do wykonania gdy ***planista przydziału*** CPU zechce dokonać kolejnego przydziału.

Kwant czasu musi być dostatecznie duży, aby między kolejnymi wymianami można **COŚ** wykonać.

◗ **Zazwyczaj proces wymieniany wraca** do **pamięci na poprzednie miejsce.**

Proces **nie może** być przesunięty w inne miejsce, jeśli wiązanie jest wykonywane podczas tłumaczenia lub ładowania.

Proces **może być** sprowadzony do innego obszaru pamięci, jeśli adresy ustala się podczas wykonania, ponieważ adresy fizyczne są obliczane na bieżąco.

◗ **Do wymiany potrzebna jest szybki dysk.**

System utrzymuje ***kolejkę procesów GOTOWYCH***, składającą się ze wszystkich procesów gotowych do działania, których **obrazy pamięci** są w pamięci **dyskowej** lub Operacyjnej.

Planista przydziału procesora decydując się wykonać proces wywołuje Ekspedytora.

Ekspedytor *(dispatcher)* sprawdza, czy następny proces z kolejki jest w **PAO**.

Jeśli nie ma tam procesu i nie ma wolnego obszaru pamięci, to ekspedytor odsyła na dysk jakiś proces przebywający w **PAO** i na jego miejsce wprowadza potrzebny proces.

Następnie uaktualnia stany rejestrów i przekazuje sterowanie do wybranego procesu.

● **Czas przełączania kontekstu w systemie z wymianą jest dość długi.**

Proces użytkownika ma rozmiar 100 KB zaś dysk twardy o szybkości przesyłania 50 MB/s .

Przesłanie 100 KB kodu procesu do/z PAO zajmuje: 100/50000 = **2 ms**

Zakładając, że ***nie trzeba przemieszczać głowic*** dysku oraz że średni czas dotarcia do sektora wynosi 8 ms, czas wymiany wyniesie 2x10 ms = **20 ms** (przesyłanie w obie stron).

◗ **Czas wymiany jest limitowany czasem przesyłania.**

Łączny czas przesyłania jest wprost proporcjonalny do wielkości wymienianej pamięci.

☛ Warto wiedzieć ***dokładnie***, ile pamięci zajmuje proces użytkownika.

Użytkownik powinien informować system o każdej zmianie zapotrzebowania na pamięć.

☛ Trzeba korzystać z funkcji **systemowych** do *zamawiania* i *zwalniania pamięci*.

➔ **Wymiana procesu wymaga pewności, że proces jest zupełnie bezczynny.**

Proces czekający z powodu operacji We/Wy może być usunięty z PAO.

Operacja We/Wy została ustawiona w kolejce, gdy urządzenie jest zajęte.

Gdyby wysłano z pamięci proces **P1** i zastąpiono go procesem **P2**, operacja We/Wy mogłaby użyć pamięci należącej obecnie do procesu **P2**.

Rozwiązania problemu:

1. **nigdy** nie wymieniać procesu, w którym trwają operacje We/Wy,
2. wykonywać operacje We/Wy tylko za pośrednictwem buforów SO.

8.1. Przydział ciągły

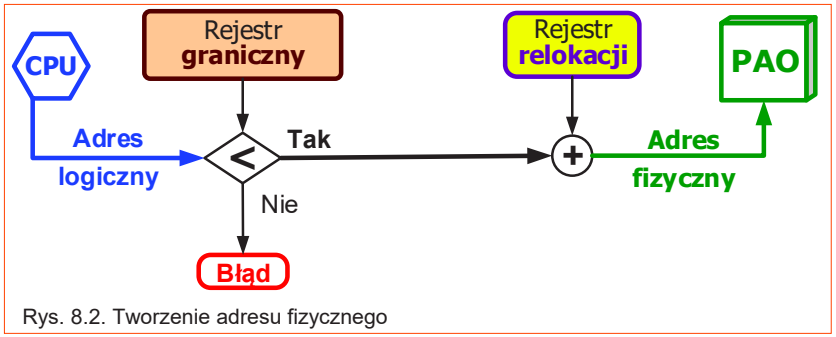
Pamięć operacyjna musi pomieścić **System Operacyjny** i **procesy użytkownika.**

❒ **Przydział pojedynczego obszaru**

Ochronę kodu i danych SO przed ingerencją procesów użytkownika, można realizować poprzez użycie **rejestru granicznego** w połączeniu z **rejestrem relokacji**.

☛ **Rejestr graniczny** zawiera **zakres** adresów **logicznych** (przestrzeń adresów)

☛ **Rejestr relokacji** zawiera wartość ***najmniejszego*** adresu **fizycznego**.



Rys. 8.2. Tworzenie adresu fizycznego

Jednostka zarządzania **PAO** przekształca dynamicznie adres logiczny **dodając** wartości **rejestru relokacji**.

Planista przydziału procesora wybiera proces do wykonania.

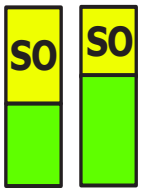
Ekspedytor ustawia wartości rejestru relokacji i rejestru granicznego - przełączanie kontekstu.

Każdy adres generowany przez CPU porównywany jest z zawartością **rejestrów**, co chroni SO, programy oraz dane innych użytkowników przed **bieżącym procesem**.

◗ **Rejestr relokacji pozwala na bieżąco śledzić rozmiar dostępnej PAO.**

Niech SO zawiera kod i obszar ***buforów modułów sterujących*** urządzeń.

Jeśli moduł sterujący urządzenia (lub inna usługa SO) nie jest często używany, to utrzymywanie jego ***kodu*** i ***danych*** w PAO jest zbędne.



Kod taki zwany **kodem przejściowym** *(transient)*, pojawia się i znika stosownie do potrzeb.

☛ Zmienia się rozmiar SO podczas wykonywania programu.

❒ **Przydzielanie wielu obszarów**

W PAO powinno pozostawać KILKA procesów użytkowych w tym samym czasie.

Prosty schemat przydziału PAO dzieli ją na obszary o **stałym rozmiarze.**
(partycjowanie)

● **M**ultiprogramming with a **F**ixed numer o **T**ask –**MFT**;
(zastosowano po raz pierwszy w IBM OS/360)

Każdy obszar zawiera **tylko jeden** proces.

Gdy program nie mieści się w partycji musi być nakładkowany.

Kiedy powstaje wolny obszar, wtedy wybiera się proces z kolejki wejściowej i wprowadza go do tego obszaru.

Proces kończąc działanie zwalnia zajmowany obszar.

● **M**ultiprogramming with a **V**ariable numer o **T**ask -**MVT**

to uogólniony schemat **MFT**, używany w środowisku wsadowym.

(partycje mają różne wielkości, ich liczba jest zmienna);

☛ SO przechowuje **Tablicę** z informacjami o zajętych i wolnych obszarach pamięci.

Wstępnie cała pamięć stanowi jeden blok i dostępna jest dla procesów użytkowych.

Przybywa nowy proces, wówczas poszukuje się dla niego **odpowiednio** dużego obszaru.

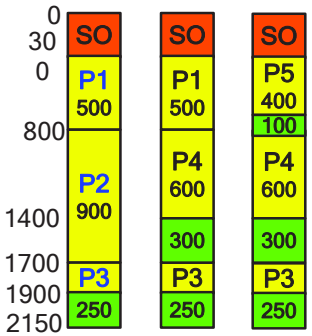
Jeśli zostanie znaleziony, to przydziela się z **niego** PAO tylko **w niezbędnej ilości**, pozostawiając resztę na przyszłe potrzeby.

☸ Dostępne jest **2150** KB PAO, przy czym SO zajmuje stale 300 KB.

Dana jest kolejka czekających **5-**ciu procesów oraz planowanie zadań metodą **FCFS**.

Można natychmiast przydzielić pamięć procesom **P1, P2, P3**.

Kolejka zadań		
Proces	PAO	Czas
P1	**500**	**10**
P2	**900**	**5**
P3	**200**	**20**
P4	600	8
P5	400	15



Rys. 8.3. Przydział pamięci

Pozostały obszar **250** KB jest za mały dla procesów czekających w kolejce.

Proces **P2** zakończy działanie i zwalnia przydzieloną mu pamięć, zaś w jego miejsce wchodzi proces **P4**.

P1 skończy pracę, zwalnia pamięć → w jego miejsce wchodzi **P5**.

Przydzielając **PAO** procesom, SO uwzględnia **zapotrzebowanie** na pamięć każdego procesu oraz ilość **wolnej** pamięci.

Proces, któremu przydzielono przestrzeń, jest wprowadzany do pamięci i zaczyna rywalizować o przydział CPU.

W każdej chwili znana jest lista rozmiarów dostępnych bloków oraz **kolejka wejściowa Procesów.**

●**Procesom przydziela się pamięć do chwili, gdy braknie bloku o** wymaganej **wielkości.**

SO może zaczekać na pojawienie się odpowiedniego bloku lub przeskoczyć pozycję w kolejce wejściowej, żeby sprawdzić, czy **inny proces** nie ma mniejszych wymagań.

● **Istnieje w PAO zbiór dziur o różnych rozmiarach, rozproszonych po całej pamięci.**

Gdy proces zamawia PAO, przeglądany jest zbiór dziur.

Jeśli dziura jest większa od procesu, to pozostała cześć wraca do zbioru dziur.

Gdy proces kończy pracę, zwalnia swój blok pamięci, który zostaje umieszczony w zbiorze dziur.

● **Jeśli nowa dziura przylega do innych dziur, to łączy się przyległe dziury.**

Należy wówczas sprawdzić, czy istnieją procesy oczekują na pamięć oraz czy zreorganizowana pamięć spełnia wymagania któregoś z tych procesów.

Dynamiczny przydział pamięci *(dynamic storage allocation)*:

strategia rozstrzygania, jak na podstawie listy wolnych **dziur** spełnić zamówienie na obszar o **zadanym** rozmiarze.

◗ Trzy strategie wyboru wolnego obszaru ze zbioru dostępnych dziur:

Pierwsze dopasowanie: przydziela się ***1-szą*** dziurę o wystarczającej wielkości.

Szukanie rozpoczyna się od początku wykazu dziur lub od miejsca, w którym je ostatnio zakończono.

Szukanie kończy napotkanie dostatecznie dużej dziury.

Najlepsze dopasowanie: przydziela się ***najmniejszą*** z dostępnych dużych dziur.

Należy przejrzeć cała listę chyba, że jest ona uporządkowana według wymiarów.

Strategia zapewnia najmniejsze pozostałości po przydziale.

Najgorsze dopasowanie: przydziela się ***największą*** dziurę.

Należy przeszukać cała listę, gdy nie jest uporządkowana według wymiarów.

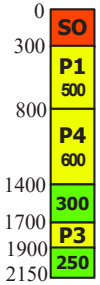
Po przydziale **pozostaje największa** dziura, która może okazać się bardziej użyteczna niż pozostałość wynikająca ze strategii **najlepszego dopasowania**.

Symulacje wykazały, że strategie wyboru **pierwszej** lub **najlepiej dopasowanej** dziury są lepsze od wyboru ***największej*** dziury zarówno pod względem zmniejszania czasu, jak i zużycia pamięci.

Zewnętrzna fragmentacja *(external fragmentation)*:

pamięć poszatkowana jest na dużą liczbę małych dziur; suma wolnych obszarów w pamięci wystarcza na spełnienie zamówienia, ale nie tworzą one spójnego obszaru.

Dokonywanie wyboru według strategii **pierwszego** dopasowania albo **najlepszego** dopasowania może wpływać na wielkość fragmentacji.



Reguła 50 procent *(50-percent rule)*:

analiza statystyczna strategii **pierwszego** dopasowania wykazała, że po przydzieleniu **N** bloków z powodu fragmentacji może ginąć **0.5N** innych bloków (nawet po optymalizacji).

Wewnętrzna fragmentacja *(internal fragmentation)*:

różnica między wielkością pamięci **zamawianej** przez proces a **przydzielonej** procesowi.

Dziura ma wielkość **12345** bajtów.

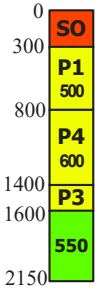
Nowy proces wymaga **12340** bajtów.

Przydzielenie zamówionego bloku zostawia **5**-cio bajtowy **nieużytek**.

Koszt trzymania informacji o takiej dziurze przekracza jej wielkość.

Zazwyczaj dołącza się bardzo małe dziury do większych przydziałów.

Należy minimalizować niepełne wykorzystanie pamięci w ramach partycji np. przypisując procesowi najmniejszą partycję, w której się jeszcze zmieści.



Upakowanie: takie przemieszczanie zawartości pamięci, aby cała wolna pamięć znalazła się w jednym bloku.

Upakowanie nie zawsze jest możliwe.

Aby procesy przemieszczone mogły pracować w nowych miejscach, należy zmienić wszystkie ich **wewnętrzne** adresy.

Jeśli ustalanie adresów jest **statyczne** i wykonywane podczas tłumaczenia, to upakowanie nie jest możliwe.

Upakowywanie możliwe tylko przy **dynamicznym wiązaniu** adresów realizowanych podczas działania procesu.

Przemieszczenie procesu sprowadza się do przesunięcia programu i danych oraz do zmiany **rejestru relokacji,** tworząc nowy adres bazowy.

◗ **Upakowaniu może towarzyszyć wymiana.**

Procesy można wysyłać z PAO na dysk i wprowadzać z powrotem w innych terminach.

Po wysłaniu procesu pamięć zostaje zwolniona i może ją zagospodarować inny proces.

◗ **Powrót procesu do pamięci może powodować pewne problemy.**

Przy **statycznym** ustalaniu adresów proces powinien być wprowadzony dokładnie w to samo miejsce, które zajmował przed wymianą.

To ograniczenie może powodować ***konieczność usunięcia*** z pamięci innego procesu w celu zwolnienia miejsca.

Przy **dynamicznym** przydziale adresów proces może zostać wysłany do innego miejsca pamięci; odnajduje się wolny blok pamięci, dokonując w razie potrzeby upakowania, i umieszcza w nim proces.

Jeśli wymiana procesów jest częścią systemu, to kod realizujący upakowywanie może być minimalny.

W przypadku upakowania, procesy przemieszczane są w PAO.

Adresy instrukcji i danych, do których proces odwołuje się zmieniają się za każdym razem, gdy proces jest ładowany do PAO lub jest w niej przemieszczany.

Dotychczas operowaliśmy pojęciami wolny lub zajęty **spójny** obszar pamięci o **stałej** lub **zmiennej** długości.

Co decyduje o zmianie długości obszarów pamięci (wolnych lub zajętych) ?

❒ **Stały rozmiar partycji PAO**

Program użytkownika operuje adresami logicznymi a nie **fizycznymi** w **PAO**.

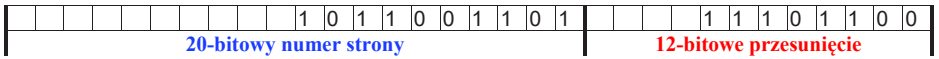
Użytkownik nie ma pojęcia, w jakich obszarach **PAO** ulokowany zostanie program.

W programie adres logiczny składa się z numeru **strony** i **przesunięcia** na stronie (przy statycznym podziale adres logiczny to położenie słowa względem początku programu).

Niech **logiczna** przestrzeń adresowa ma rozmiar 2^m, zaś strona 2^n [bajtów lub słów], wtedy

m – n bardziej znaczących bitów adresu logicznego wskazuje **numer strony**,

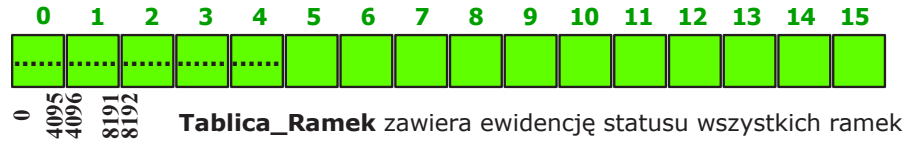
n mniej znaczących bitów ***przesuniecie*** na stronie.



Niech strona 2^{12} =**4096** B;

logiczna przestrzeń adresowa 2^{32} czyli 2^{20} **= 1048576 stron.**

Pamięć fizyczna **RAM** jest podzielona na obszary o **stałej** długości - **ramki** np. 4 KB.



Tablica_Ramek zawiera ewidencję statusu wszystkich ramek

Wybierając komórkę w PAO istotny jest **numer ramki** i pozycja komórki względem **jej** początku.

➔ **Numer ramki** jednoznacznie określa adres początkowy komórki w PAO.

Rozmiar strony (a także ramki) określa sprzęt, i jest **potęgą 2** ∈ (512 B, 16 MB).

Niech program na dysku, składa się z 6-ciu stron.

Gdy ma być załadowany do *PAO*, *SO* wyszukuje w niej 6 wolnych ramek i wczytuje do nich 6 stron procesu.

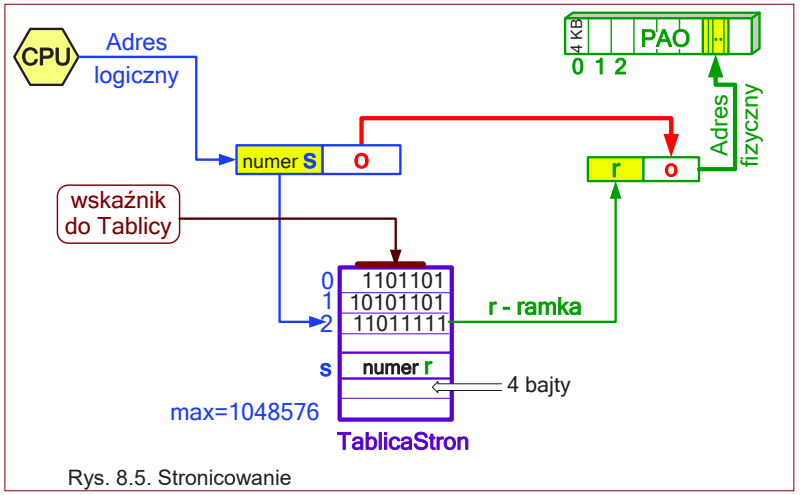
Two gigabytes is equal to 2,000 megabytes.
However, GB has historically been used in the fields of computer science and information technology to refer to **1,073,741,824** bytes

8.2. Stronicowanie

Stronicowanie *(paging)* umożliwia przydział **dowolnie dostępnych** miejsc w pamięci fizycznej, dopuszcza ***nieciągłość*** logiczną przestrzeni adresowej procesu.

Pamięć **logiczna** dzieli się na bloki **STAŁEJ** długości zwane **stronami** *(pages)*.

Pamięć **fizyczna** PAO dzieli się na bloki stałej długości zwane **RAMKAMI** *(frames)*.



Rys. 8.5. Stronicowanie

➔ **Pamięć dyskową** dzieli się na bloki (strony) o stałej długości i rozmiarze ramki PAO ➔

Wykonanie procesu wprowadza jego **strony** z pamięci dyskowej do **ramek** w PAO.

Adres tworzony przez CPU składa się z **dwu części**:

-numer strony s jest ***indeksem*** w **TablicyStron[s]**

-odległość na **stronie o** *(offset)* lub przemieszczenie.

TablicaStron *(page table)* zawiera ***adresy bazowe*** **stron** danego procesu w **PAO**.

Pozycja w TablicyStron ma typowo 4 bajty, więc może wskazywać na jedną z 2^{32} ramek w **PAO**.

Dla **ramek** o rozmiarze 4 KB (2^{12}) system może adresować 2^{44} bajtów czyli 16 TB **PAO.**

➔ **Proces trafia do wykonania**

➔ **jego strony ładowane** są do **dostępnych RAMEK**.

Każda wirtualna **strona** procesu odwzorowana jest przez jedną **ramkę** w **PAO**.

Pierwsza strona procesu jest ładowana do ***przydzielonej* ramki**, a numer tej ramki wpisuje się do **TablicyStron danego procesu**.

Następną **stronę** wprowadzi się do innej **ramki** itd..

➔ Każdy proces ma **własną TablicęStron** ➔

☛ **Ramka przechowuje stronę procesu.**

Każdy wpis w **TablicyStron** zawiera numer ramki w PAO, o ile ramka znajduje się w PAO.

Teoretycznie efektywniejsze są małe rozmiary strony czyli **dużo ramek w PAO**

Z każdą stroną wiążą się koszty przechowywania wpisu w TablicyStron;
(duże strony, mała Tablica)

Wydajność operacji We/Wy rośnie ze zwiększaniem się porcji przesyłanych danych.

Niech **strona** i **ramki** mają rozmiar **4 KB** zaś pamięci fizyczna 32 KB (8 ramek).

Odwzorowanie pamięci **logicznej** (wirtualnej) - **strony**

0 1 2 3	4 5 6 7	8 9				
A l f a	B e t a	M a r a	b u t y			
Strona 0	Strona 1	Strona 2	Strona 3	Strona4	Strona 5	Strona 6

Odwzorowanie pamięci **fizycznej** - **ramki**

M a r a	b u t y			A l f a	B e t a		
0 ramka **0**	4 ramka **1**	8 **2**	12 **3**	16 **4**	20 **5**	24 **6**	28 **7**

Tablica stron

s	Numer ramki
0	5
1	6
2	1
3	2
4	
5	
6	

Adres logiczny **0** oznacza odległość **0** na stronie **0**.

TablicaStron[0] = **5** → co oznacza, że strona **0** znajduje się w ramce **5**.

Logiczny adres **0** odwzorowany na adres fizyczny: **5** x **4** + **0** = **20** (5 x 4096).

Logiczny adres **3** (strona **0**, odległość **3**,) odwzorowany jest na adres fizyczny: **5** x **4** + **3** = **23**

Logiczny adres **6** (strona **1**, odległość **2**). **Jaki jest adres fizyczny ?**

TablicaStron[1] = **6** → co oznacza, że strona **1** znajduje się w ramce **6**.

Logiczny adres **6** odwzorowany jest na adres fizyczny: **6** x **4** + **2** = **26** (6 x 4096 + 2048).

System operacyjny musi znać szczegóły przydzielania PAO

-jaka jest ogólna liczba ramek,

-które ramki są **przydzielone** oraz które są **wolne**.

Tablica_Ramek (*frame table)* jest strukturą danych, przechowującą te informacje.

W **Tablicy_Ramek** dla każdej fizycznej ramki danej strony przydzielono jeden rekord.

Każda pozycja **Tablicy_Ramek** zawiera informacje:

czy **ramka** jest **wolna,** czy **zajęta;**

a jeśli zajęta, to, przez którą **stronę** i **jaki proces**.

- **Stronicowanie eliminuje zewnętrzną fragmentację.**

Każda **wolna ramka** może być przydzielona potrzebującemu jej procesowi.

Może wystąpić fragmentacja wewnętrzna.

Ramki mają **ustaloną** wielkość.

Jeśli wielkość procesu nie jest wielokrotnością rozmiaru ramek, to ostatnia przydzielona ramka nie będzie w pełni wykorzystana:

n stron + **jedno słowo** = **n + 1** ramek

- **Stronicowanie rozdziela pamięć oglądaną przez użytkownika** od **pamięci fizycznej**. .

SO nadzoruje i ukrywa przed użytkownikiem odwzorowanie adresów logicznych na fizyczne.

- **Proces użytkownika** z definicji **nie może sięgnąć do pamięci, której nie jest właścicielem.**

Nie może zaadresować PAO leżące poza **jego TablicąStron**.

Tablica zawiera odwzorowania stron należących ***tylko*** do określonego procesu.

SO musi mieć pewność, że:

- procesy użytkowników działają w przestrzeni użytkowników,

- adresy logiczne są odwzorowywane na **adresy fizyczne**.

Jeśli użytkownik wywoła funkcję systemową (np. operację We/Wy) i przekaże w parametrze adres (np. bufor), to **adres ten** musi zostać odwzorowany na poprawny **adres fizyczny**.

Stronicowanie wydłuża czas przełączania kontekstu.

SO utrzymuje **kopię** **TablicyStron** każdego użytkownika, (podobnie jak kopie licznika rozkazów i zawartości rejestrów).

8.2.1. Budowa tablicy stron

TablicaStron zazwyczaj przydzielana jest do każdego procesu.

➔ W bloku kontrolnym procesu przechowuje się ***wskaźnik*** do TablicyStron.

Ekspedytor rozpoczyna proces, i określa wartości ***sprzętowej tablicy*** stron wg przechowywanej w pamięci **TablicyStron** Procesu.

TablicaStron może być **zbiorem rejestrów** o dużej szybkości działania ⇨ mała Tablica..

Ekspedytor określa zawartość tych rejestrów.

Rozkazy modyfikujące rejestry TablicyStron wykonywane są w trybie uprzywilejowanym

Dużą **TablicęStron** przechowuje się w PAO ➔

jej położenie wskazuje **Rejestr Bazowy TablicyStron** (***P**age-**T**able **B**ase **R**egister* - **PTBR**).

➔ Czas dostępu do komórki pamięci procesu ulega dwukrotnemu spowolnieniu.

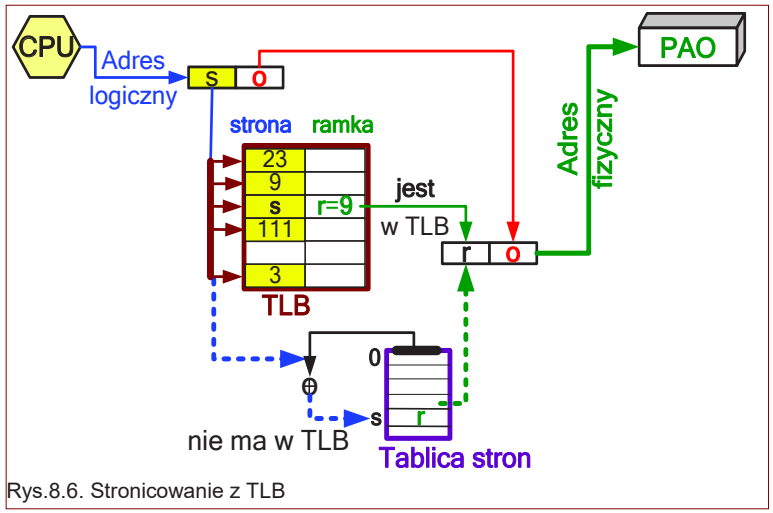
Wymagane są ***dwa*** kontakty z pamięcią: -dostęp do TablicyStron,
-dostęp do danego bajta.

Dotarcie do komórki **x** wymaga wpierw dostępu do **TablicyStron**, aby otrzymać numer **ramki**.
Numer **ramki** i **odległość** na stronie utworzą **aktualny adres** komórki **x**.

❒ **Rejestr asocjacyjny** *(associative registers)*

szybka sprzętowa pamięć podręczna, zwana też **Buforem Translacji Adresów Stron** (***T**ranslation **L**ook-aside **B**uffers* - **TLB**).

Każdy rejestr **asocjacyjny** składa się z dwu części: **klucza** i **wartości**.
Porównanie obiektu z kluczami w **TLB** odbywa się **równocześnie** dla **wszystkich kluczy**.
Zgodność obiektu z kluczem udostępnia pole wartości.



Rys.8.6. Stronicowanie z TLB

Rejestr asocjacyjny zawiera kilka wpisów z **T**ablicy**S**tron.

TLB zawiera numery **stron** i odpowiadające im **ramki**.

Numer **strony adresu logicznego** porównywany jest ze zbiorem **rejestrów asocjacyjnych**.

Jeśli **numer strony** zostanie w **TLB** odnaleziony, to dostęp do numeru ramki jest natychmiastowy ⇨ dostęp do **PAO**.

Jeśli **numeru strony nie ma** w **TLB**, następuje odwołanie do miejsca w **PAO**, gdzie przechowywana jest **TablicaStron**.

Dodatkowo dołącza się nowy numer strony i ramkę do **rejestrów asocjacyjnych**, aby przy następnym odwołaniu numery te były dostępne w **TLB**.

Jeśli rejestry asocjacyjne **są pełne**, to SO wybiera któryś z nich do zastąpienia wartości.

Po każdym wyborze **nowej TablicyStron** (np. każdorazowo po przełączaniu kontekstu), **rejestry asocjacyjne** zostają opróżnione.

Inaczej w TLB zostałyby stare wpisy zawierające poprawne adresy wirtualne, lecz z błędnymi adresami fizycznymi, pozostałymi po poprzednim procesie.

❒ **Współczynnik trafień** *(hit ratio):* procent numerów stron odnajdywanych w rejestrach asocjacyjnych.

Niech współczynnik trafień wynosi **80%**.

☛ Niech przeglądanie rejestrów asocjacyjnych zabiera **20** ns, dostęp do PAO **100** ns, a numer strony ***jest*** w rejestrach, wówczas odwzorowywany dostęp do **PAO** zajmuje **120** ns.

☛ Jeśli **nie** powiedzie się odnalezienie numeru strony w rejestrach asocjacyjnych (20 ns), to:

SO sięga do pamięci po **TablicęStron** i numer **ramki** (**100** ns),

po czym odwołuje się do właściwego słowa w pamięci (**100** ns)

co łącznie daje **220** ns.

Efektywny czas dostępu do PAO *(effective memory-access time)*:

uwzględnia do każdego z dwóch w/wym. przypadków- **wagi** wynikające z prawdopodobieństwa ich wystąpienia.

efektywny czas dostępu = **0.80** x 120 + **0.20** x 220 = **140** ns

Dla współczynnika trafień 98%:

efektywny czas dostępu = 0,98 x 120 + 0,02 x 220 = **122** ns

Współczynnik trafień zależny jest od liczby rejestrów asocjacyjnych.

◗ **Procesy rzadko działają** w **całej przestrzeni swoich adresów.**

Sprzętowy **rejestr długości TablicyStron** (***P**age-**T**able **L**ength **R**egister* - **PTLR**) umożliwia posługiwanie się **rozmiarem** **T**ablicy**S**tron.

Każdy **adres logiczny** jest porównywany z zawartością **PTLR** w celu sprawdzenia, czy należy do ***przedziału*** dozwolonego dla procesu.

❒ Ochrona

Bity ochrony przypisane **każdej** ramce, realizują ochronę pamięci w systemie stronicowanym.

Bit określa stronę, jako dostępną do czytania i pisania albo ***wyłącznie*** do czytania.

Każde odwołanie do pamięci przechodzi przez TablicęStron w celu odnalezienia numeru ramki.

W **czasie obliczania** adresu fizycznego, można sprawdzać **bity ochrony** w celu zapobieżenia próbom pisania na stronie dostępnej tylko do czytania.

Próba pisania na stronie przeznaczonej wyłącznie do czytania spowoduje przejście do SO.

Bit ochrony pozwala SO na wychwycenie prób niedopuszczalnych rodzajów dostępów.

Bit poprawności *(valid-invalid bit):* wpis w TablicyStron jest uzupełniany dodatkowo o bit.

Stan ***„poprawny"*** oznacza, że **strona**, z którą jest on związany, znajduje się w logicznej przestrzeni adresowej procesu ➔ strona jest dozwolona.

Stan ***„niepoprawny"*** oznaczy, że strona nie należy do logicznej przestrzeni adresowej procesu.

Niedozwolone adresy są wychwytywane za pomocą sprawdzania stanu **bitu poprawności.**

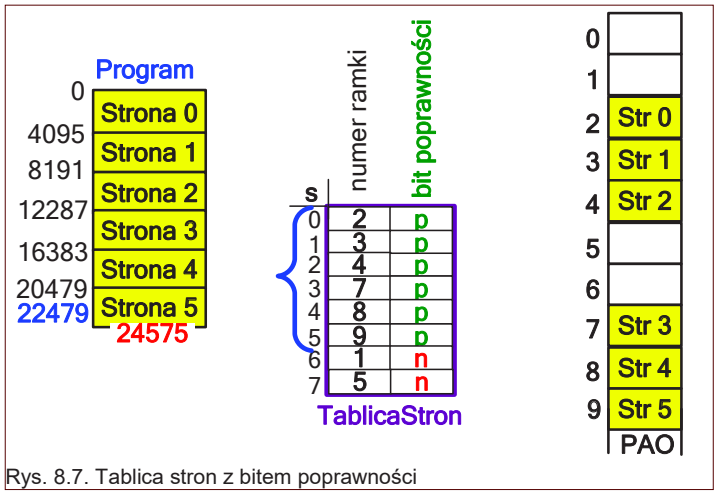
SO nadaje wartość **bitowi poprawności** w odniesieniu do **każdej strony**, zezwalając lub zakazując na korzystanie ze strony.

✱ Dany jest system z 16-bitową przestrzenią adresową, zaś strona ma rozmiar 4 KB.

Przestrzeń adresowa programu: **0** do **22479**.

Adresy należące do stron **0, 1, 2, 3, 4** i **5** są odwzorowywane za pomocą TablicyStron.

Przy próbie utworzenia adresu odnoszącego się do stron **6** lub **7** wartość **bitu poprawności** jest równa ***„niepoprawne"*** i nastąpi przejście do SO.



Rys. 8.7. Tablica stron z bitem poprawności

Teoretycznie każde odwołanie sięgające powyżej wartości **22479** jest niedozwolone.

Odwołanie do strony **5** jest poprawne, co umożliwia dostęp do adresu **24575**.

Problem ten wynika z rozmiaru strony (4KB) i odzwierciedla wewnętrzną fragmentację występującą w stronicowaniu.

8.2.2. Stronicowanie wielopoziomowe

System o 32-bitowej przestrzeni adresowej ma 2^{20} (1048576) **stron** o rozmiarze 4 KB.

W takim środowisku sama **TablicaStron** zajmuje dużo **ciągłego obszaru** PAO.

TablicaStron może zawierać milion pozycji (każda **4**B).

Każdy proces może potrzebować **4** MB(2^{22}) PAO na ***samą*** **TablicęStron**.

➔ Zatem **podział TablicyStron** na **części** ➔

Niech **TablicaStron** składa się z **1024** (2^{10}) **podtablic** z rozmiarze 4 KB (2^{12}) każda.

32-bitowy adres logiczny dzieli się na **20**-bitowy **numer strony** i **12**-bitową **odległość** na stronie.

❒ Schemat stronicowania dwupoziomowego

Katalog Stron i **TablicaStron**.

20-bitowy numer strony = **10**-bitowy **Katalog Stron** plus **10**-bitowa **TablicaStron**

➔ Każdy proces pamięta **własną** wartość **Adresu Bazowego Katalogu Stron** ➔

Pierwsze **10-bitów** indeksuje **KatalogStron** wybierając adres **podtablicy** z TablicyStron.

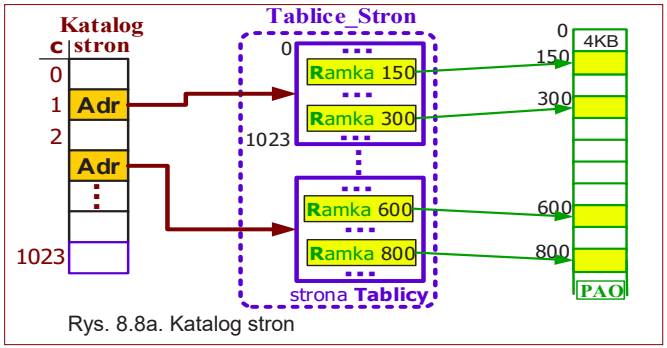
Drugie **10-bitów** indeksuje wybraną **podtablicę** z **TablicyStron**, aby ustalić numer **ramki** w PAO.

Adres logiczny przyjmuje postać:

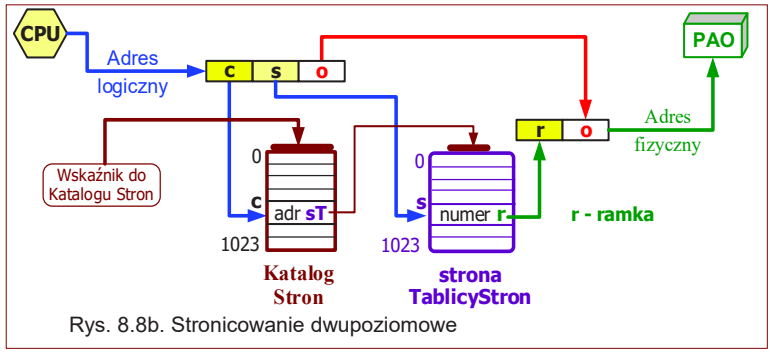
c -indeks w **Katalogu Stron** (do podtablic TablicyStron),

s -przesunięcie na stronie**Tablicy** (indeks do ramki).

numer strony		odległość na stronie
Katalog **c**	Strona **s**	**o**
10 bitów	**10** bitów	**12** bitów



Rys. 8.8a. Katalog stron



Rys. 8.8b. Stronicowanie dwupoziomowe

Negatywny wpływ stronicowania wielopoziomowego na wydajność systemu

Przekształcenie adresu logicznego na fizyczny może wymagać **4-ch** dostępów do pamięci, co zwiększa łączny czas realizację jednego dostępu do PAO **5-ciokrotnie**.

Poprawę przynosi zastosowanie szybkiej pamięci podręcznej.

Dla współczynnika trafień **98**%:

efektywny czas dostępu = **0,98** x **120** + 0,02 x **520** = **128** ns

Przy dostępie do PAO **100** ns, dodatkowe poziomy tablic wydłużają **efektywny czas dostępu** o 28%.

❒ Schemat stronicowania trzypoziomowego

Zawiera trzy struktury tablic.

Katalog stron:

Aktywny proces ma jeden katalog stron zazwyczaj o wielkości jednej strony.

Zapis w tym katalogu wskazuje na jedną stronę **Pośledniego katalogu stron**.

Katalog stron aktywnego procesu musi znajdować się w pamięci głównej.

Pośredni katalog stron:

Katalog ten może obejmować wiele stron.

Każdy zapis w nim wskazuje jedną stronę TablicyStron.

Tablica stron:

Tablica stron też może obejmować wiele stron.

Każdy zapis TablicyStron odnosi się do jednej strony **logicznej** danego procesu.

Adres logiczny składa się z czterech pól:

s1 - indeks katalogu stron.

s2 - indeks pośredniego katalogu stron.

s3 - indeks tablicy stron.

o - przesunięcie w ramach wybranej strony pamięci.

numer strony			odległość
s1	s2	s3	o

Procesor Intel 80386 zawierał sprzętowy mechanizm obsługi **dwupoziomowego** schematu stronicowania.

Procesor Motorola 68030 zawierał schemat stronicowania czteropoziomowego.

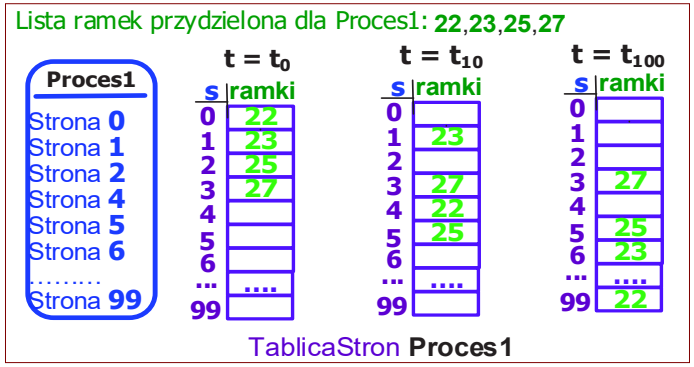
❒ Haszowana Tablica Stron

Każdy proces posiada **własną** **TabliceStron**, która może zawierać dużo pozycji.

TablicaStron zawiera jedną pozycję (adres ramki) dla każdej strony wirtualnej procesu.

Jest **ona** uporządkowana wg **adresów wirtualnych**, więc adres fizyczny **ramki PAO** jest dostępny natychmiast.

TabliceStron używają dużo PAO tylko do pokazania, jak użytkowany jest inny obszar PAO.



TablicaStron w systemach 64-bitowych: 64 bity adresują 16EB (czyli 17 179 869 184 GB)!

TablicaStron dla takiej przestrzeni adresowej (strony 4kB) zajmowałaby ok. 32PB !

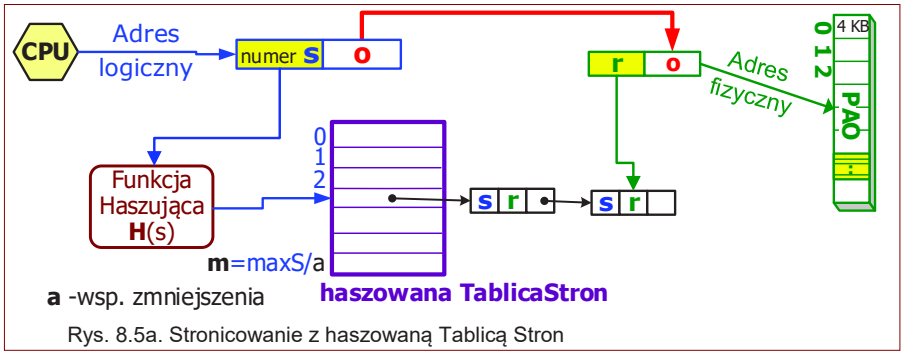
Liczba poziomów stronicowania w nieznacznym stopniu rozwiązuje problem.

Problem rozwiązuje **haszowana** TablicaStron (*hashed page table*).

☛ Wartością haszowaną jest **numer strony** logicznej (wirtualnej).

☛ Poprzez dobór funkcji haszującej można utworzyć TabliceStron dowolnie **małą**.

Każde pole tablicy z haszowaniem zawiera listę elementów określających to samo miejsce, powstałe w wyniku działania funkcji haszującej ➔ lista obsługuje kolizje.



Rys. 8.5a. Stronicowanie z haszowaną Tablicą Stron

Wynik funkcji haszującej (adres logiczny) porównywany jest z polem **s** **1-go** elementu listy.

Jeżeli wystąpi zgodność, to otrzymujemy adres **ramki**.

Jeżeli **nie** wystąpi zgodność to przeglądane są następne elementy listy.

Funkcja haszującą może mieć postać **H(s) = s mod m** , gdzie jest rozmiarem Tablicy z haszowaniem.

8.2.3. Odwrócona tablica stron

Odwrócona Tablica Stron zawiera tyle pozycji, ile ramek pamięci fizycznej jest faktycznie zainstalowanych w komputerze.

OdwróconaTablicaStron ma po jednej pozycji dla każdej **ramki PAO** odwzorowującej **stronę**.

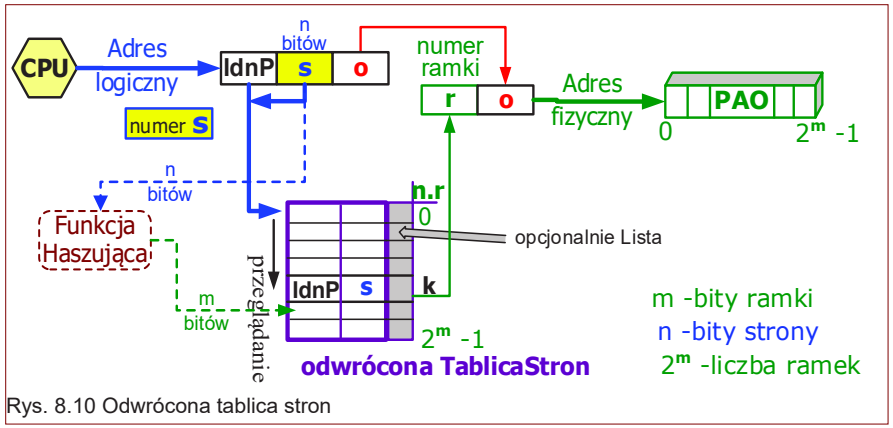
➔ W **OTS** indeks Tablicy jest numerem **ramki,** zaś **TS indeksowało** się na podstawie wirtualnego **numeru strony**.

W systemie istnieje tylko jedna OdwróconaTablicaStron.

Każda pozycja zawiera: -**Identyfikator procesu**, do którego strona należy,
-**adres logiczny** strony.

☛ Wpis w **O**dwróconej **T**ablicy**S**tron: **< Identyfikator_procesu, numer_strony,** [Lista] **>**

Struktura adresu logicznego: **< IdnProcesu, numer_strony, odległość >**



Rys. 8.10 Odwrócona tablica stron

Odwołanie do pamięci, przekazuje ***część*** adresu logicznego:

<IdnProcesu, numer_strony> do podsystemu pamięci.

Następuje przeszukanie **OTS** w celu dopasowania adresu.

Jeśli dopasowanie powiedzie się (spełni **k-ty** element OTS), to **indeks** OST tworzy adres fizyczny: **<k, odległość>**.

☛ Taka struktura zmniejsza rozmiar PAO potrzebnej do pamiętania **TablicStron** procesów.

Zwiększa czas przeszukania **OTS** przy odwołaniu do strony, gdyż jest **ona** uporządkowana według ***adresów fizycznych***.

Przeglądanie dotyczy adresów wirtualnych, zatem należy przeszukać ją w całości.

Zazwyczaj **n > m**, stąd funkcja haszująca odwzorowuje **n**-bitowy numer strony na **m**-bitową wartość

Czas przeszukiwania poprawiają rejestry **pamięci asocjacyjnej**, w których przechowuje się **ostatnio** zlokalizowane wpisy.

☛ Rejestry przeglądane są **przed** zaglądaniem do Tablicy.

❒ Strony dzielone

Zaletą stronicowania jest możliwość dzielenia wspólnego kodu.

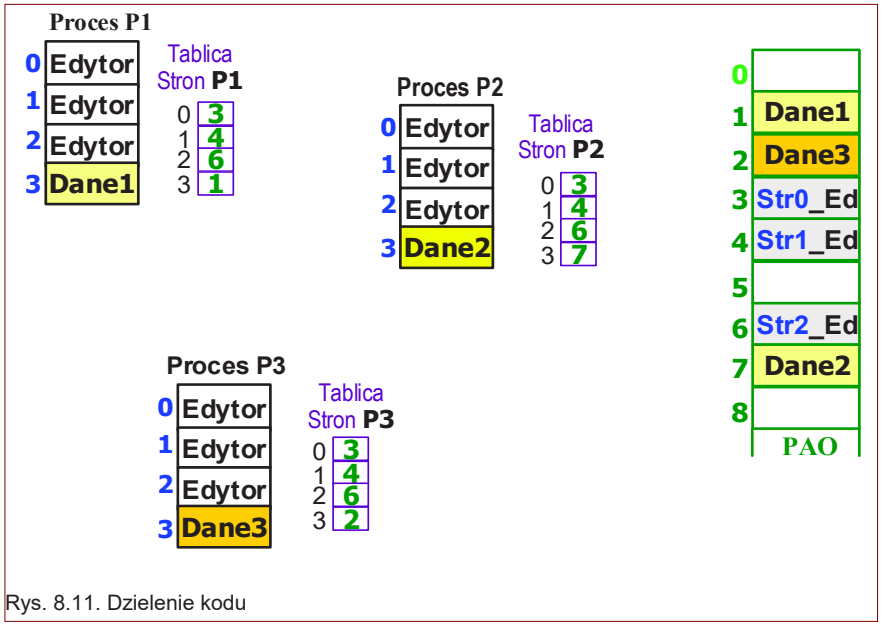
Kod wznawiany: nie modyfikuje sam siebie, nic nie może go zmienić podczas wykonania.

Kodem wznawianym można się dzielić.

Kilka procesów może wykonywać ten sam kod w tym samym czasie.

Edytor trzystronicowy składa się ze **150** KB kodu i **50** KB obszaru danych.

Do obsługi 3 użytkowników, korzystających z edytora potrzeba **600** KB PAO.



Rys. 8.11. Dzielenie kodu

Każdy proces ma swoją kopię rejestrów i obszar danych do przechowywania wyników.

Dane dla **dwu różnych** procesów będą **różne**.

W pamięci fizycznej wystarczy przechowywać **jedną kopię** edytora.

TablicaStron każdego użytkownika odwzorowuje adresy rozkazów na tę samą fizyczną kopię edytora, natomiast **strony *danych*** są odwzorowywane na różne **ramki**.

Do obsługi 3 użytkowników wystarczy jedna kopia edytora i 3 kopie danych, łącznie 300 KB.

8.3. Segmentacja

Jak użytkownik widzi pamięć operacyjną ?

Jak myśli pisząc poważny program ?

Widzi zbiór funkcji lub modułów z wyróżnionym programem głównym oraz strukturami danych.
Każdy moduł lub obiekt danych jest identyfikowany za pomocą nazwy i ma swoją wielkość.
Elementy wewnątrz segmentu identyfikuje ich odległość od początku segmentu:
1-sza instrukcja programu, 3-cia pozycja w tablicy symboli, 5-ta instrukcja Funkcji(), itd.

Segmentacja *(segmentation)*: schemat zarządzania pamięcią, w którym przestrzeń adresów logicznych jest zbiorem segmentów.

Stronicowanie: użytkownik określa pojedynczy adres, dzielony następnie przez sprzęt na numer strony i odległość - w sposób niewidoczny dla programisty.

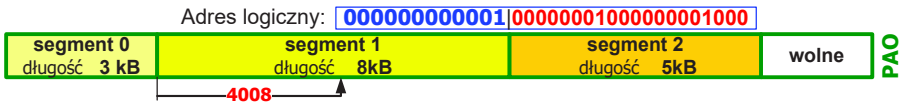
Segmenty programu mogą być **różnej** długości.

Każdy segment ma **nazwę** i **długość**.

Adres określany jest za pomocą dwu wielkości: -**nazwa segmentu**, -**odległość**.

Adres logiczny tworzy para: **<numer_segmentu, odległość>**

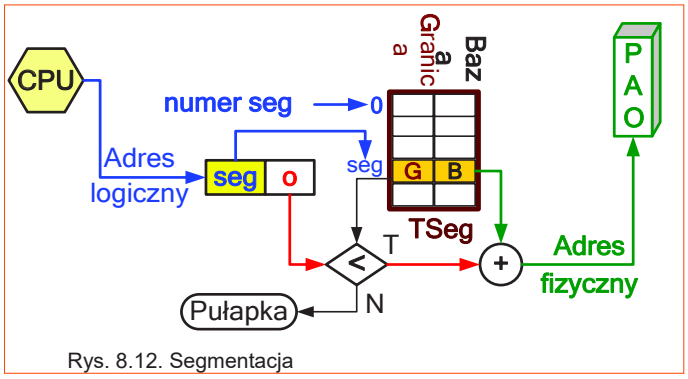
Kompilatory automatycznie konstruują segmenty odpowiadające programowi wejściowemu.
Kompilator może tworzyć segmenty dla: -zmiennych globalnych,
-stosu wywołań funkcji,
-kodu funkcji, -lokalnych zmiennych funkcji.



Użytkownik odwołuję się do obiektów w programie za pomocą adresu dwuwymiarowego.

☛ Rozpoczęcie procesu, ładuje jego segmenty do PAO i tworzona jest **TablicaSegmentów**.

TSeg odwzorowuje dwuwymiarowe adresy użytkownika na jednowymiarowe adresy fizyczne.



Rys. 8.12. Segmentacja

Struktura **T**ablicy **Seg**mentów (zbiór par):

Granica_segmentu: jego **długość**;

Baza_segmentu: adres fizyczny początku segmentu.

Adres logiczny: -numeru segmentu **seg,**
-odległości w segmencie **o.**

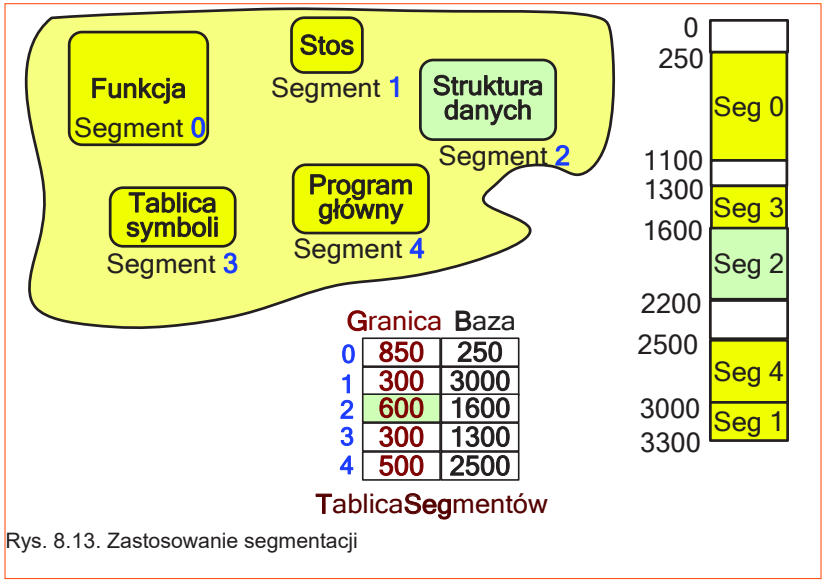
Numer segmentu jest ***indeksem*** **T**ablicy**Seg**mentów**[seg]**

Odległość o adresu logicznego ∈ (**0**, **Granica_segmentu**).

Jeśli tak nie jest, to pułapka w SO sygnalizuje: *<adres logiczny poza końcem segmentu>*

Fizyczny adres bajta = Baza segmentu + odległość o

Dane: 5 segmentów numerowanych od 0 do 4.
Segmenty przechowane są w pamięci fizycznej.
Tablica segmentów ma oddzielne pozycje dla każdego segmentu:
-długość segmentu (Granica),
-adres początkowy segmentu w PAO (Baza).



Rys. 8.13. Zastosowanie segmentacji

Segment 2 ma długość **600** B i zaczyna się od adresu 1600.

Odwołanie do **66** bajta **Segmentu 2** jest odwzorowywane na adres: 1600 + 66 = 1666.

Odwołanie do bajta **888 Segmentu 0** spowoduje awaryjne przejście do systemu operacyjnego, ponieważ segment ten ma **długości 850** B.

8.3.1. Implementacja Tablicy Segmentów

Tablica segmentów może być umieszczona w: -szybkich rejestrach,
-pamięci operacyjnej.

Ulokowanie Tablicy Segmentów w **rejestrach** zapewnia szybki dostęp, gdyż dodawanie do bazy i porównywanie z wartością graniczną mogą być wykonywane jednocześnie.

Dużą Tablicę Segmentów przechowuje się w **PAO** (za mało rejestrów).

Rejestr bazowy TablicySegmentów (***Segment-Table Base Register*** - **STBR**) wskazuje na **T**ablicę**S**egmentów.

Rejestr długości TablicySegmentów (***Segment-Table Length Register*** - STLR) stosowany jest ze względu na dużą zmienność liczby **segmentów** w programie.

Dany jest adres logiczny (**segment, odległość**).

-Najpierw sprawdza się poprawność numeru segmentu: **segment < STLR.**

-Następnie dodaje się numer **seg** do rejestru bazowego **T**ablicy**S**eg: **seg + STBR**;
wynikiem jest adres lokalizujący w PAO odpowiednią pozycję w **TSeg**.

Czyta się zawartość wybranej pozycji **TSeg** a następnie:

-sprawdza się, czy odległość nie przekracza długości segmentu,

-oblicza się adres fizyczny bajta jako sumę bazy segmentu i odległości.

Tak realizowane odwzorowanie wymaga **dwu odwołań** do PAO dla każdego adresu logicznego.

Aby przyspieszyć dostęp do PAO stosuje się **zbiór rejestrów asocjacyjnych**, w których przechowuje się ostatnio używane pozycje **T**ablicy**Seg**mentów.

❒ **Ochrona** i **wspólne użytkowanie**

Segmenty mogą zawierać rozkazy lub dane.

Segmenty rozkazów można zdefiniować, jako przeznaczone tylko do **Czytania** lub **Wykonywania**.

Sprzęt odwzorowujący pamięć będzie sprawdzał **bity ochrony** związane z każdą pozycją Tablicy Segmentów, nie dopuszczając do nieuprawnionych dostępów do pamięci.

☛ Po umieszczeniu **tablicy danych** w osobnym **segmencie**, sprzęt zarządzający pamięcią będzie automatycznie sprawdzał, czy indeksy tej tablicy są poprawne.

Zaletą segmentacji jest **dzielenie kodu** lub **danych.**

Każdy proces ma swoją **T**ablicę**Seg**mentów, której ekspedytor używa do zdefiniowania sprzętowej Tablicy Segmentów w chwili przydzielania CPU danemu procesowi.

Dzielenie segmentów występuje wtedy, gdy wpisy w tablicach dwu różnych procesów wskazują na to samo w PAO.
Dzielenie występuje na poziomie segmentów.

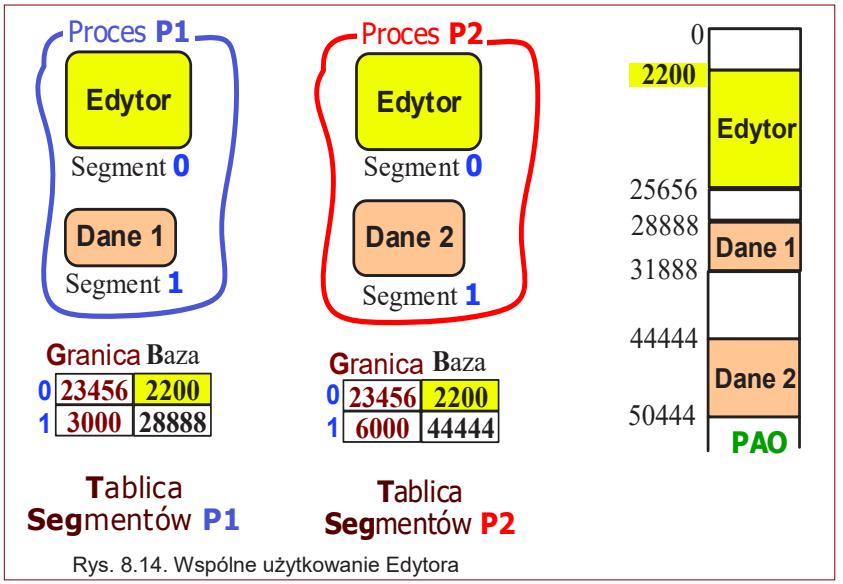
☛ Dowolna informacja może być dzielona, **jeśli została zdefiniowana** jako **segment**.

Dzieleniu może podlegać kilka segmentów ⇒ program wielosegmentowy może być dzielony.

Użytkowanie edytora tekstu w systemie przez dwóch użytkowników.

☛ Wystarczy jedna kopia edytora.

Każdemu użytkownikowi trzeba zapewnić osobne segmenty na zmienne lokalne.



Rys. 8.14. Wspólne użytkowanie Edytora

Wspólne pakiety podprogramów mogą być dzielone między wielu użytkowników, jeśli zdefiniuje się je, jako segmenty dzielone.

Dwa programy mogą używać tego samego podprogramu ***Sqrt***, do czego wystarcza w pamięci tylko jedna jego kopia.

Segmenty kodu mogą zawierać odwołania do adresów w ich obrębie.

8.4. Uwagi

Segmenty mają zmienne długości.

Planista przydziela pamięć wszystkim segmentom programu użytkownika.

Przydział segmentów ma charakter przydziału dynamicznego, zwykle realizowany za pomocą algorytmu najlepszego lub pierwszego dopasowania.

Segmentacja umożliwia programiście traktowanie pamięci tak, jakby składała się z wielu przestrzeni adresowych.

1. Upraszcza obsługę rosnących struktur danych.
   Struktury danych można przypisywać do oddzielnego segmentu i SO będzie zmieniał jego **wielkość** w miarę potrzeby.
   Jeżeli segment, który należy zwiększyć, znajduje się w PAO i jest tam za mało miejsca, wówczas SO może go przesunąć do większego obszaru w PAO lub przesłać na dysk.
2. Umożliwia niezależne modyfikowanie i rekompilację oprogramowania bez ponownego konsolidowania i ładowania całego pakietu programów (wiele segmentów).
3. Ułatwia procesom współdzielenie.
   Można utworzyć segment ze wspólnym blokiem danych, do którego mogą odwoływać się inne procesy.
4. Ułatwia realizację ochrony pamięci.
   Segment można tak zbudować, aby zawierał struktury danych, w których programista może łatwo umieszczać stosowne uprawnienia.

● **Segmentacja może powodować zewnętrzną fragmentację.**

➔ Każdy blok wolnej pamięci może być za **mały**, by pomieścić **cały** segment.

W tym przypadku proces może czekać na zwiększenie obszaru pamięci lub można zastosować upakowanie w celu utworzenia większej dziury.

Jeśli planista przydziału CPU musi opóźniać proces z powodu kłopotów z przydziałem pamięci, to może on przeskoczyć miejsce w kolejce do CPU w poszukiwaniu procesu mniejszego, nawet o niższym priorytecie.

Czy **zewnętrzną fragmentacją** jest sprawą poważną w technice segmentowanie ?

Zależy to od segmentu **średniego** rozmiaru, jego mały rozmiar zwiększa prawdopodobieństwo, że zewnętrzna fragmentacja będzie mała.

Duże segmenty można potraktować, jako osobne procesy.

Każde słowo (w skrajnym przypadku) można traktować jak osobnym segment.

Podejście takie usuwa zewnętrzną fragmentację całkowicie; jednak **każdy bajt** potrzebowałby rejestru ***bazowego*** do wykonywania przemieszczeń.

Stronicowanie i segmentacja mają zalety i wady.

Stronicowanie jest ***transparentne*** dla programisty i umożliwia efektywne wykorzystanie PAO.

Porcje danych przesyłanych do PAO mają stałą wielkość, możliwe jest opracowanie optymalnych **algorytmów zarządzania**, uwzględniających zachowanie się programów.

Segmentacja jest ***widoczna*** dla programisty, i umożliwia obsługę rosnących struktur danych, modularność, obsługę współdzielenia oraz realizację ochrony.

❒ Segmentacja i stronicowanie

W systemie stronicowania/segmentacji przestrzeń adresowa użytkownika podzielona jest na pewną liczbę (wg. uznania użytkownika) **segmentów**, a każdy z nich składa się ze **stron** o ustalonej wielkości, równej wielkości **ramki PAO**.

Jeżeli **segment** jest mniejszy niż **strona**, to zajmuje jedną stronę.

Dla **programisty** adres logiczny to: **n**umeru **segmentu** i **p**rzesunięcie w **segmencie**.

Dla **systemu** **p**rzesunięcie w segmencie składa się z:

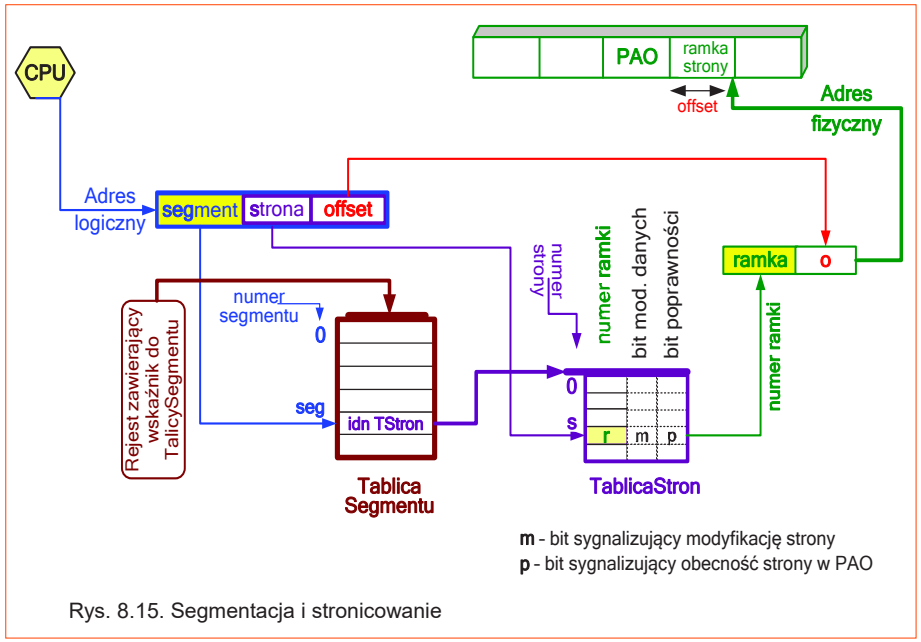
-numeru **strony**,

-przesunięcia strony w danym segmencie.

Po uruchomieniu procesu system przechowuje wskaźnik do **T**ablicy**Seg**mentu.

Fragment adresu logicznego wykorzystywany jest przez CPU do indeksowania **T**ablicy**Seg**mentu, zawierającej informacje pozwalające na lokalizację **T**ablicy**S**tron segmentu.

➔ Każdy segment ma własną **T**ablicę**S**tron.



Rys. 8.15. Segmentacja i stronicowanie

ANEX 8.1. Wirtualna Przestrzeń Adresowa Procesu w systemie Windows

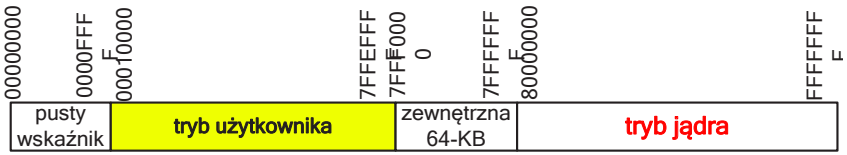
Dla procesów 32-bitowych przestrzeń adresowa wynosi 4 GB ⇐ 0x00000000 ÷ 0xFFFFFFFF

Dla procesów 64-bitowych jest to przestrzeń 16 EB (egzabajtów) (18 446 744 073 709 551 616 różnych wartości).

8.4.1. Strefy wirtualnej przestrzeni adresowej

Wirtualna przestrzeń adresowa każdego procesu dzieli się na **strefy**.

Podział ten wynika z wewnętrznej implementacji systemu operacyjnego.



Rys. 8.15. Strefy wirtualnej przestrzeni adresowej w Windows

❒ **Strefa przypisania** do **pustego wskaźnika** 0x00000000 ÷ 0x0000FFFF

Jeśli wątek procesu próbuje odczytać/zapisać z tej strefy, CPU zgłasza naruszenie praw dostępu.

Ochrona tej strefy pomaga wykrywać przypisania do pustego wskaźnika (NULL), gdyż kontrola błędów w programach C++ nie zawsze jest dokładna.

```
double *pA = new double; *pA = -12.34;
```

Jeśli *new* nie może znaleźć żądanej ilości pamięci, zwraca NULL.

Podany kod nie sprawdza tej możliwości i próbuje dostać się do adresu pamięci 0x00000000. Jest to strefa przestrzeni adresowej spoza dozwolonego zakresu, i proces zostaje zakończony.

❒ **Strefa trybu użytkownika** 0x00010000 ÷ 0x**7**FFEFFFF

Strefa prywatnej (nie dzielonej) przestrzeni adresowej procesu.

Żaden proces nie może korzystać z danych znajdujących się w tej strefie innego procesu.

Aplikacje przechowują tutaj istotne dane procesu ⇨ co zwiększa stabilność systemu.

W Windows 2000 do tego obszaru ładowane są wszystkie pliki .exe i moduły .DLL. .

Proces korzysta z mniej niż połowy przydzielonego mu obszaru (górny zakres**7**FF**E**FFFF)

System Windows 2000 Advanced Server zwiększa strefę trybu użytkownika do 3 GB.

Aby aplikacje mogły bezpiecznie używać 3-GB strefy trybu użytkownika należy dołączyć klucz **/3GB** w systemowym pliku `BOOT.INI`; nowy zakres wynosi 0x00010000 ÷ 0x**B**FFEFFFF

Użycie klucza /3GB, ogranicza obszar jądra do 1 GB.

W trybie jądra nie ma wirtualnej przestrzeni adresowej do zarządzania pamięcią.

❒ **Strefa zewnętrzna 64-kilobajtowa** 0x7FFF0000 ÷ 0x7FFFFFFF

Próba sięgnięcia do pamięci w tej strefie (czytanie/pisanie) powoduje naruszenie praw dostępu.

Strefę zarezerwowano w celu ułatwienia implementacji systemu operacyjnego.

Niech **aplikacja użytkownika** wywołuje poniższą funkcję systemową:

WriteProcessMemory(GetCurrentProcess(), 0x7FFEEE90, **Buf**, sizeof(Buf), **NumBytesWritten**);

Do funkcji systemowej przekazywany jest adres i długość bloku pamięci.

Przed swoim wykonaniem funkcja sprawdza poprawność tego bloku.

Dla funkcji tej klasy docelowy obszar pamięci sprawdzany jest przez kod trybu jądra, który może sięgać do strefy trybu jądra (adresy powyżej 0x80000000).

Jeśli istnieje pamięć pod adresem 0x80000000, powyższe wywołanie zakończy się zapisaniem danych w pamięci, która powinna być dostępna tylko dla kodu w trybie jądra.

Aby zapobiec temu, strefę tą umieszczono poza dozwolonymi granicami, więc próba sięgnięcia do pamięci w niej powoduje **naruszenie praw dostępu**.

❒ **Strefa trybu jądra** 0x80000000 ÷ 0xFFFFFFFF

Miejsce rezydowania kodu systemu operacyjnego.

Do niej ładowany jest kod szeregowania wątków, zarządzania pamięcią, obsługi plików i sieci oraz wszystkie sterowniki urządzeń.

Zawartość tej strefy jest współużytkowana przez wszystkie procesy.

Cały obszar jest chroniony.