

Jądro systemu na podstawie Linuxa

Aleksander Morgała
Bartłomiej Długosz

Programowanie
niskopoziomowe

Plan prezentacji:

1. Szybkie wprowadzenie do jądra systemu
2. Zarządzanie pamięcią
3. Zarządzanie procesami
4. Wirtualny system plików
5. Trochę o kompilacji jądra i modułach
6. System calls

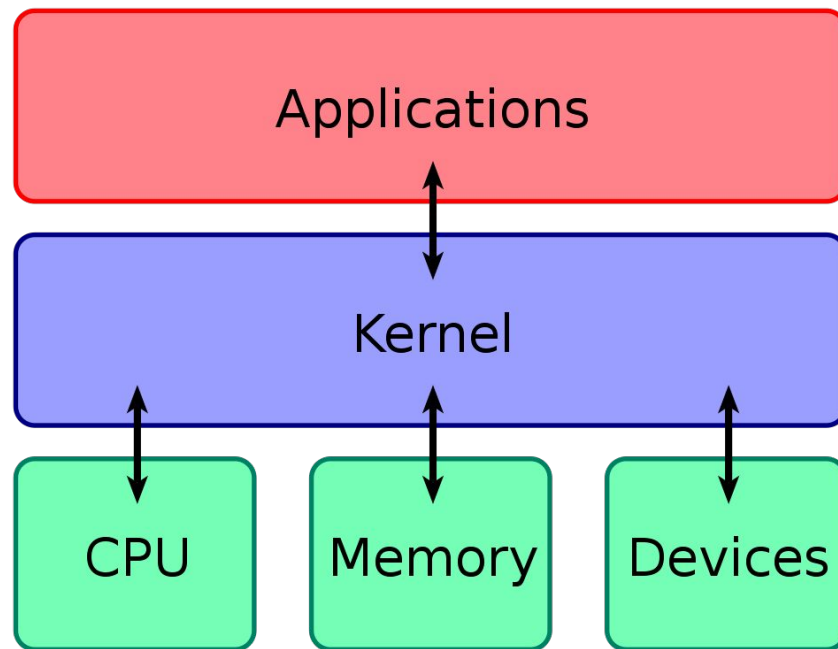
Wprowadzenie do jądra systemu

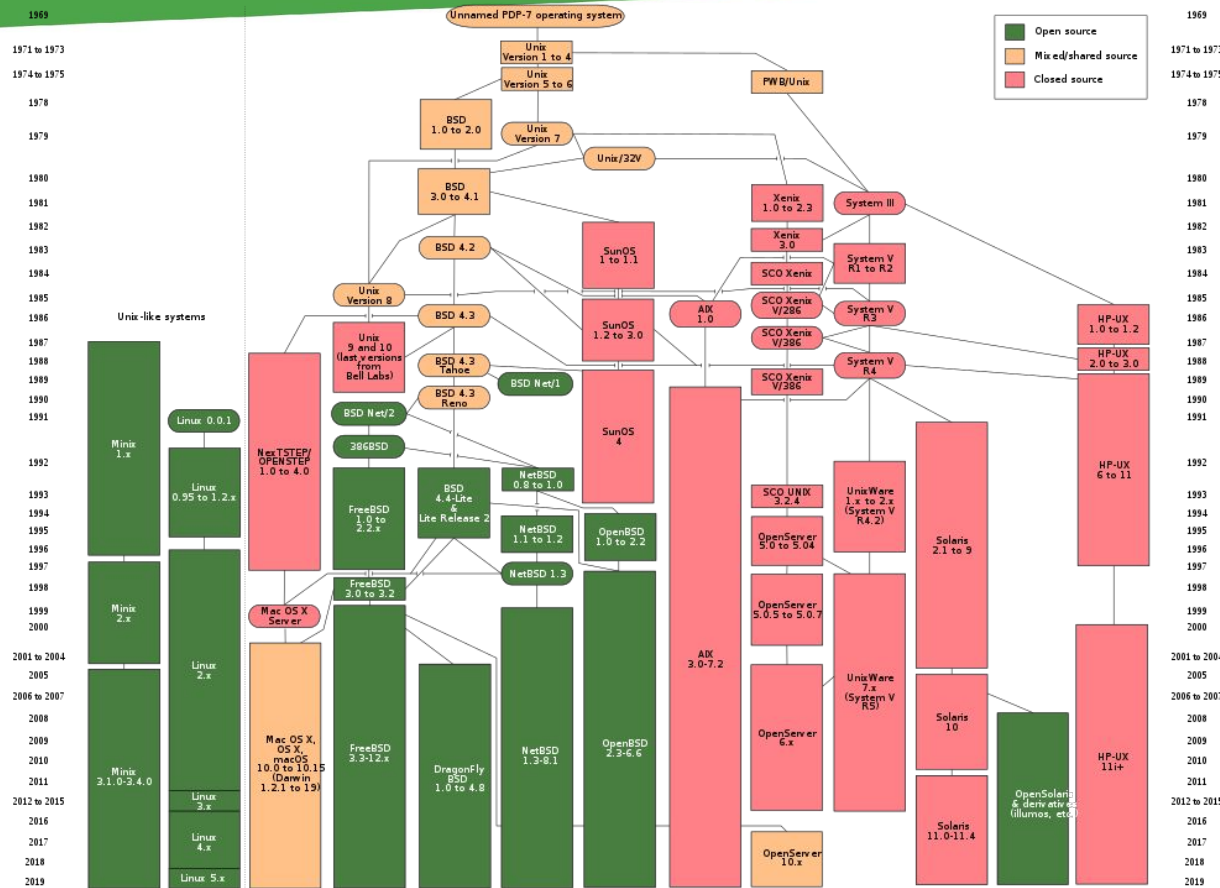
Szybkie wprowadzenie do jądra systemu



- Co nazywamy jądrem systemu
- Jakie są funkcjonalności jądra
- Rodzaje jąder

Co nazywamy jądrem systemu?





Cechy jądra(linux)

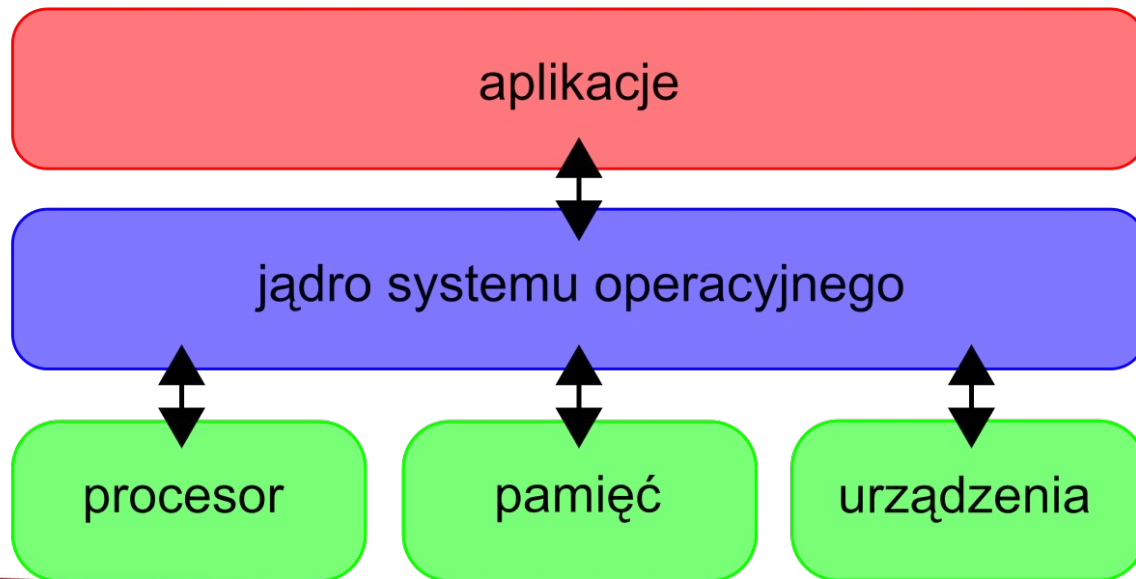
- wieloprocusowość
- wielowątkowość
- wielobieżność
- skalowalność
- wywłaszczalność

Funkcjonalności jądra

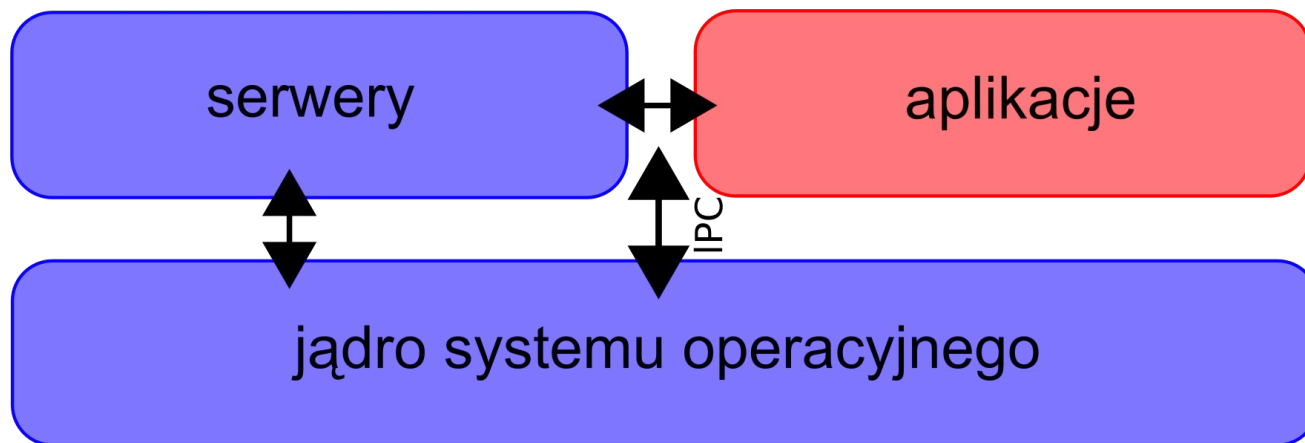
- Zarządzanie pamięcią
- Zarządzanie plikami
- Zarządzanie procesami
- Obsługa urządzeń
- Obsługa połączeń sieciowych

Rodzaje jąder

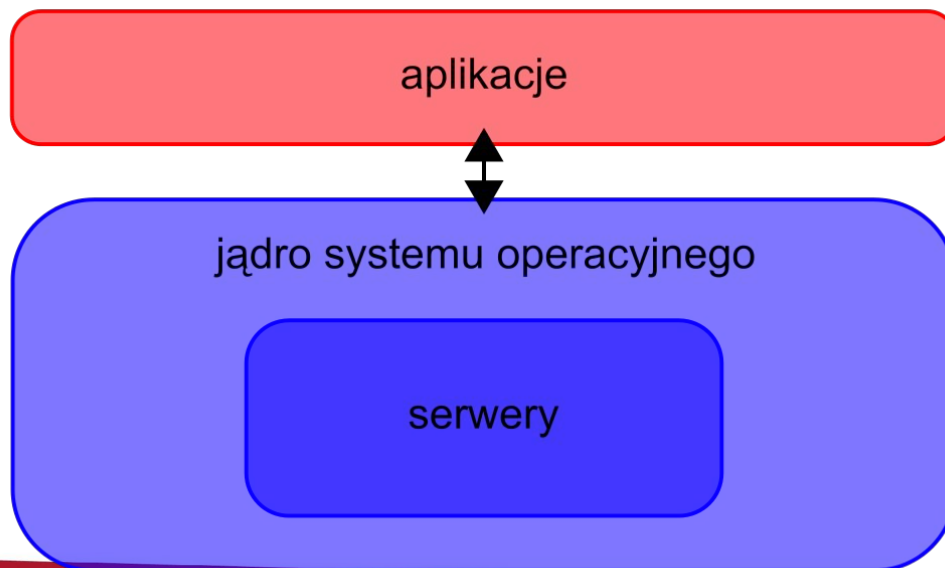
1. Monolitowe - Jądro wszystkie zadania wykonuje bezpośrednio.
np. Linux, FreeBSD



2. Mikrojądro - Jądro korzysta z “serwerów” do komunikacji z hardwarem.



3. Jądro hybrydowe - Połączenie jądra monolithowego z mikrojądrem. W jądro są wbudowane serwery.



2. Zarządzanie pamięcią.

- a) pamięć fizyczna i logiczna
- b) stronicowanie i tabele stronicowe
- c) slab allocator & buddy system

2. Zarządzanie pamięcią.

- a) Pamięć fizyczna i logiczna
 - System operacyjny “widzi” pamięć jako jeden, ogromny blok.
 - Pamięć w systemie operacyjnym takim jak Linux organizowana jest poprzez **stronicowanie** (o tym za chwilę).

0x0b05
0x0b04
0x0b03
0x0b02
0x0b01

2. Zarządzanie pamięcią.

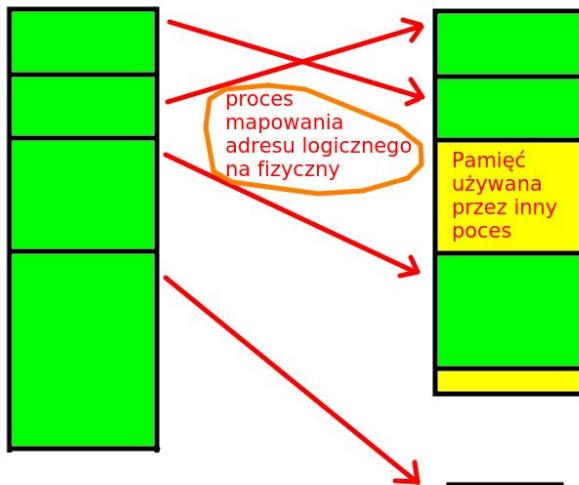
- a) **Pamięć fizyczna i logiczna**

- Każdy proces używa innego widoku na pamięć systemu – używa on własnej pamięci **logicznej**.
- Każda operacja związana z pamięcią logiczną – odniesieniem się do danego adresu w przestrzeni adresowej - wiąże się ze znalezieniem odpowiadającemu mu adresowi w pamięci fizycznej.
- **Pamięć wirtualna** jest sposobem (techniką) radzenia sobie
- z wykonywaniem wielu procesów równocześnie.

2. Zarządzanie pamięcią.

Pamięć logiczna
(dla każdego z procesów)

Pamięć fizyczna



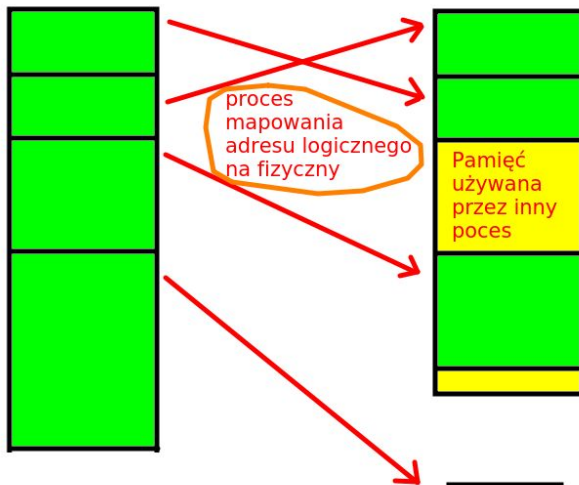
Dysk

- **a) Pamięć fizyczna i logiczna**
- Każda operacja związana z pamięcią logiczną – odniesieniem się do danego adresu w przestrzeni adresowej - wiąże się ze znalezieniem odpowiadającemu mu adresowi pamięci fizycznej.

2. Zarządzanie pamięcią.

Pamięć logiczna
(dla każdego z procesów)

Pamięć fizyczna



Dysk

• a) Pamięć fizyczna i logiczna

• A po co tak ?

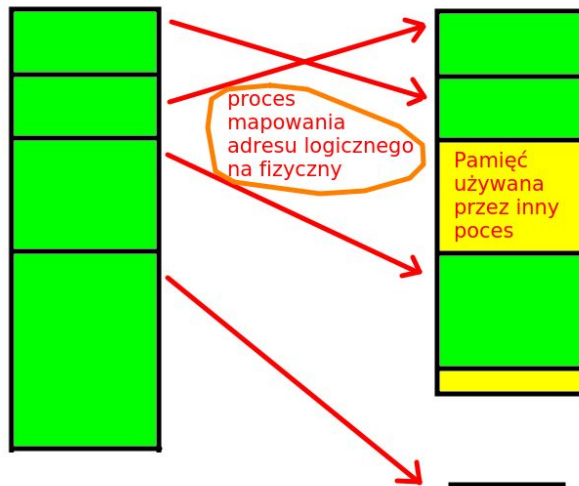
- W dzisiejszych komputerach wiele procesów wykonuje się równocześnie, przez co istnieje ryzyko nadpisywania obszarów pamięci – wykonywanie wielu procesów byłoby wręcz niemożliwe!

- Dzięki mapowaniu adresu logicznego na adres fizyczny mamy pewność, że nie odwołamy się do już zajętego obszaru pamięci lub obszaru

2. Zarządzanie pamięcią.

Pamięć logiczna
(dla każdego z procesów)

Pamięć fizyczna



Dysk

- **a) Pamięć fizyczna i logiczna**
- Nie ma też znaczenia, gdzie w rzeczywistej pamięci znajduje się adres, do którego odwołuje się proces – z jego punktu widzenia operuje na swojej przestrzeni adresowej.
- Z perspektywy procesu jest on jedynym wykonującym się w danym momencie.

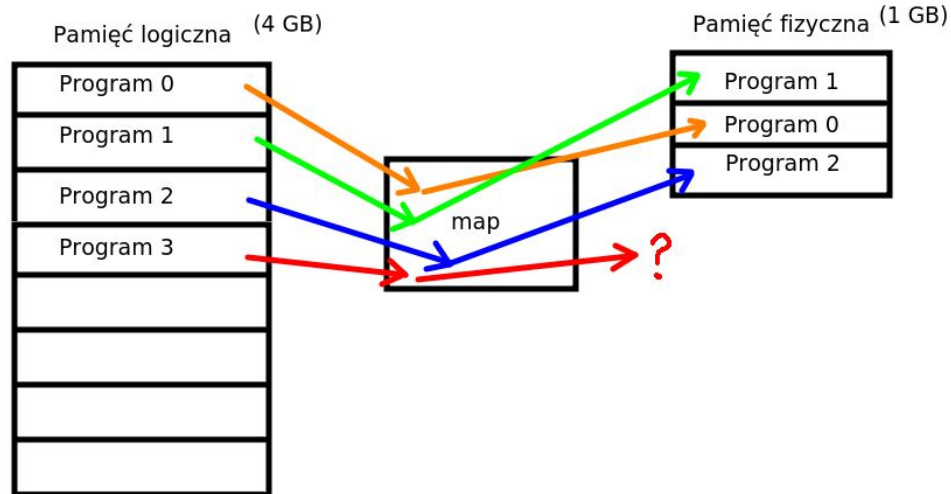
Programowanie
niskopoziomowe

2. Zarządzanie pamięcią.

- a) Pamięć fizyczna i logiczna

- Problem:**

- Brak wystarczającej pamięci fizycznej lub nieciągłość przestrzeni adresowej.



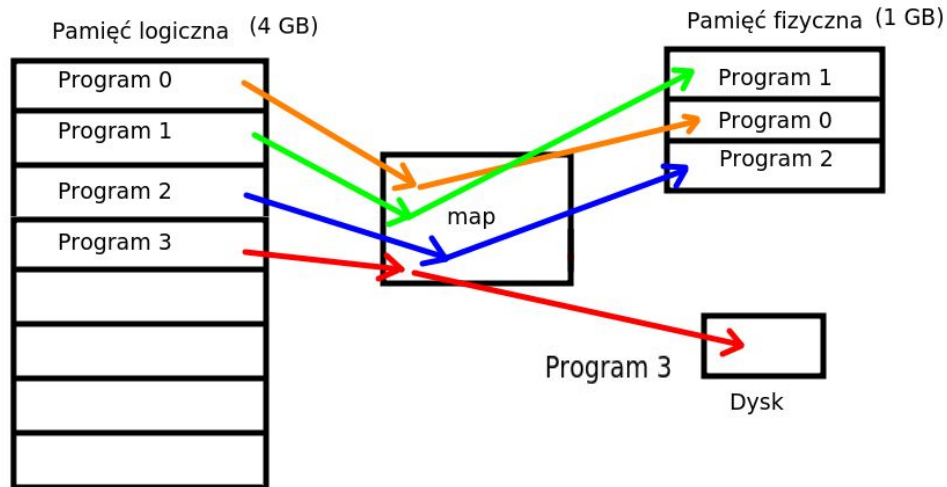
2. Zarządzanie pamięcią.

- a) Pamięć fizyczna i logiczna

- Rozwiązanie:**

- Zmapowanie części przestrzeni adresowej programu na dysk.

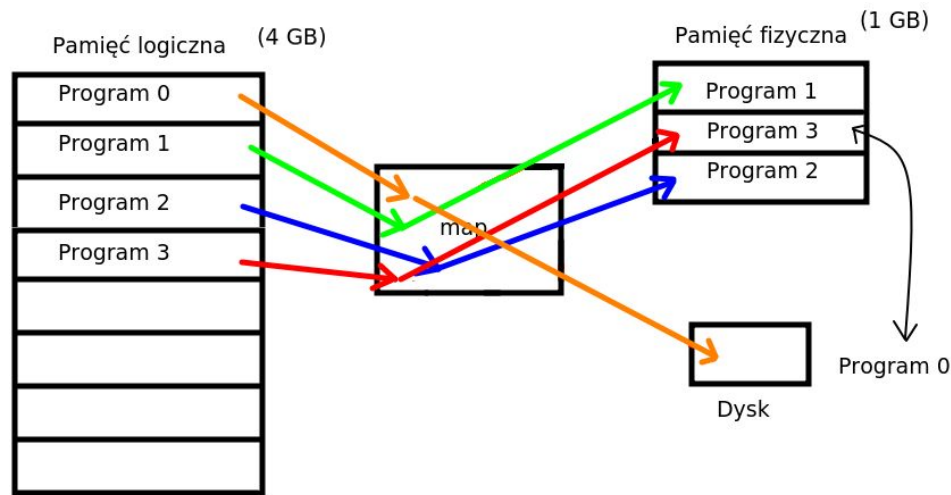
W ten sposób, gdy dostęp do niej będzie potrzebny - przeniesiona będzie ona do pamięci RAM.



2. Zarządzanie pamięcią.

• a) Pamięć fizyczna i logiczna

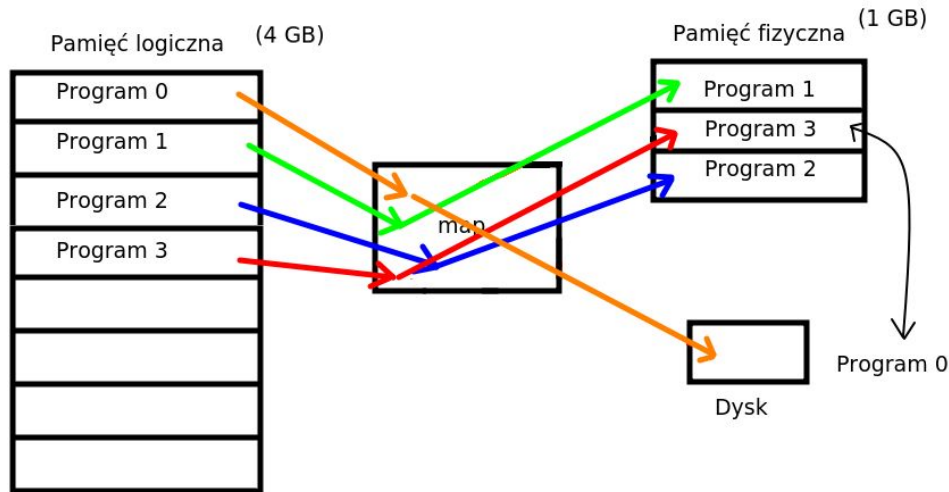
- Wiąże się to z pewnymi czynnościami, a mianowicie:
- “Przenoszony” zostaje jeden z adresów na dysk.
- Tym samym zwalnia on miejsce dla adresu, który obecnie jest umieszczany w pamięci fizycznej.



2. Zarządzanie pamięcią.

• a) Pamięć fizyczna i logiczna

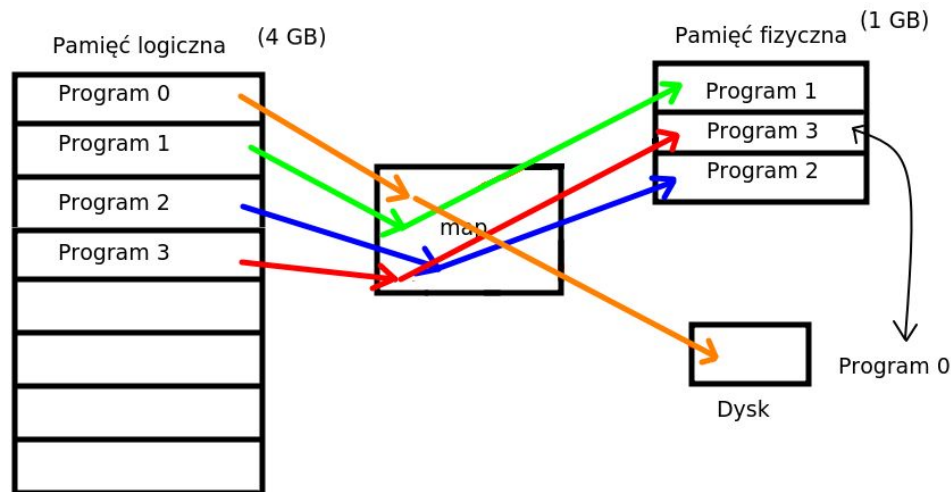
- Z punktu widzenia procesu posiada on do dyspozycji bardzo dużą, jednorodną i szybką pamięć logiczną.
- Nie ma on pojęcia, gdzie faktycznie jest przechowywany: czy to na dysku, czy dyskietce czy pamięci fizycznej RAM.



2. Zarządzanie pamięcią.

- a) Pamięć fizyczna i logiczna

- Jest to rozwiązanie problemu fragmentacji danych. I na przykład program 0 może być rozłożony na kilka części w rzeczywistej pamięci fizycznej.



2. Zarządzanie pamięcią.

- a) Pamięć fizyczna i logiczna
- Zatem:
- **Pamięć fizyczna** – realny, rzeczywisty byt w naszym komputerze.
-
- **Pamięć logiczna** - jest to pamięć na użytek procesów generowana przez jądro systemu. Czyli dla każdego procesu jest widoczna jako pamięć zaczynająca się od 0 do N-tego adresu, natomiast nie jest to rzeczywisty adres w pamięci fizycznej.

2. Zarządzanie pamięcią.

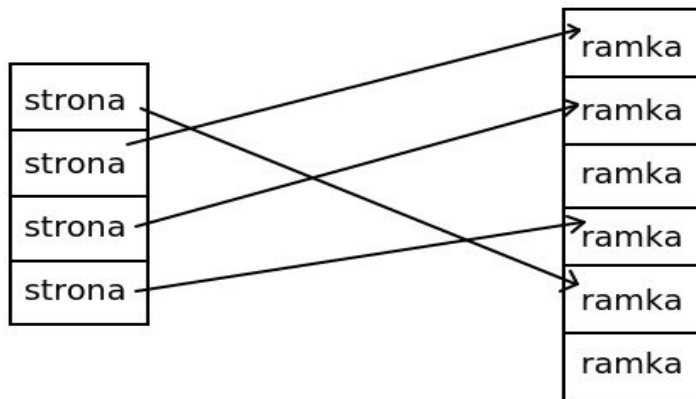
- b) Stronicowanie i tabele stronicowe
- **Stronicowanie** jest sposobem rozwiązania problemu nieciągłości rozmieszczenia logicznej przestrzeni adresowej procesu w pamięci fizycznej.
- Polega ono na podziale pamięci na spójne bloki ustalonej wielkości.
- Takie bloki pamięci logicznej nazywa się **stronami** i z reguły mają one rozmiar **4 kb**.
- Natomiast bloki pamięci fizycznej nazywa się **ramkami**
-

2. Zarządzanie pamięcią.

- b) Stronicowanie i tabele stronicowe
- **Tabela stronicowa** to struktura, która konwertuje adres logiczny na adres fizyczny.
- Znajduje się ona w pamięci **RAM** (w przestrzeni jądra).
- **Tabele stronicowe** zawierają w sobie **pozycje wejściowe** (page table entries).
- Indeksami tabeli stronicowej są logiczne numery **stron**, a wartościami adresy **ramek**.

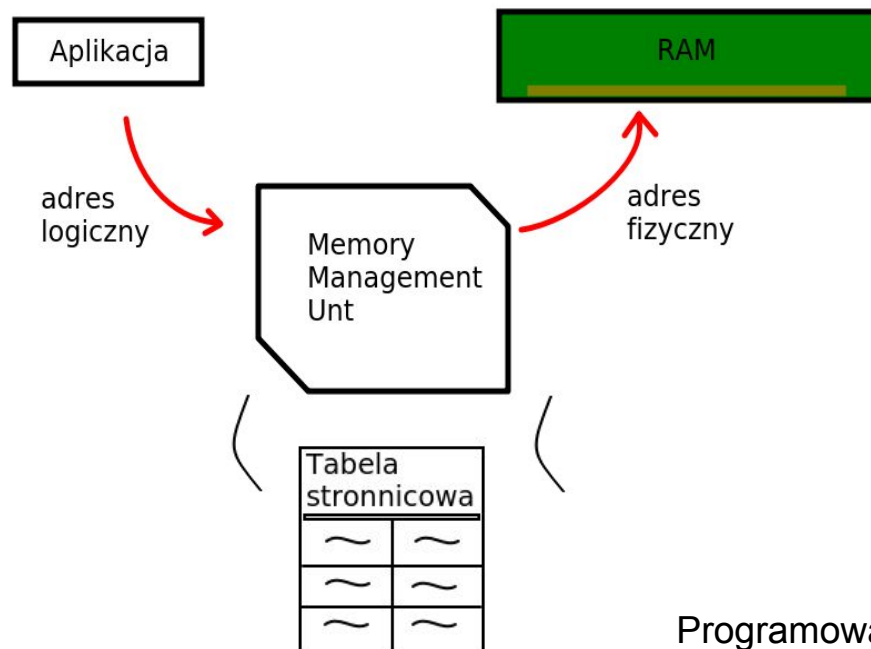
2. Zarządzanie pamięcią.

- b) Stronicowanie I tabele stronicowe
- Każda **strona** ma swój odpowiednik w postaci **ramki**.
- **Strony** są poukładane w ciągły obszar pamięci, jednak nie implikuje to ciągłości **ramek**.



2. Zarządzanie pamięcią.

- b) Stronicowanie i tabele stronicowe
- **Memory Management Unit (MMU)**
- tłumaczy ona adres logiczny na adres fizyczny.
- **MMU** operuje na adresach stron i mapuje na odpowiedni adres fizyczny.



2. Zarządzanie pamięcią.

- b) Stronicowanie i tabele stronicowe
- **Adresowanie**
- Każdy proces korzysta ze swojej pamięci logicznej – należy to przekształcić w adres fizyczny i dokonać odpowiedniej operacji.
- Tłumaczenie odbywa się w oparciu o tablicę stronicową danego procesu.
- Sprzętowe wykonywanie tłumaczenia adresów logicznych na fizyczne jest dużo szybsze niż tłumaczenie programowe – należy tylko podmienić część bitów

Tabela stronicowa

Virtual Address	Physical Address
512	12
786	4
1024	2
...	...
...	...
...	...

2. Zarządzanie pamięcią.

- **b) Stronicowanie i tabele stronicowe**

- **Jak odbywa się adresowanie ?**

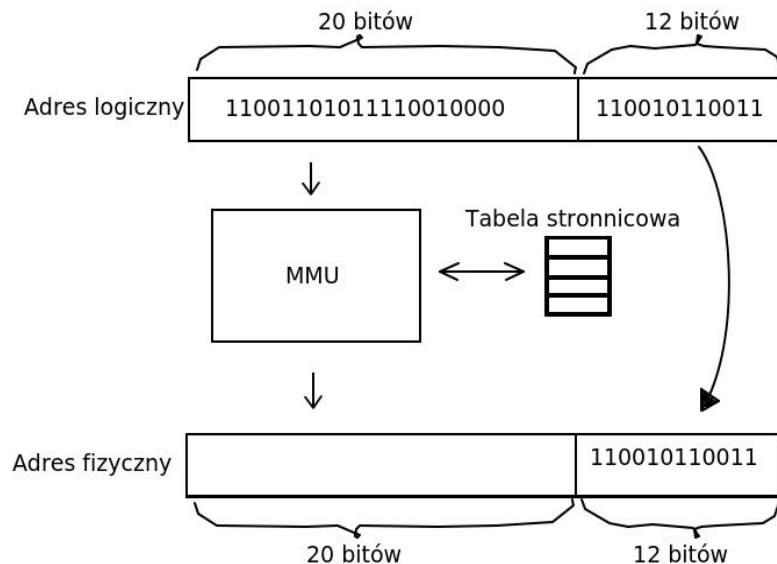
• Adres logiczny zajmuje 32 bity, z czego 20 bitów oznacza numer strony, natomiast 12 bitów oznacza przesunięcie (offset) na wskazanej stronie.

Tabela stronicowa

Virtual Address	Physical Address
512	12
786	4
1024	2
...	...
...	...
...	...

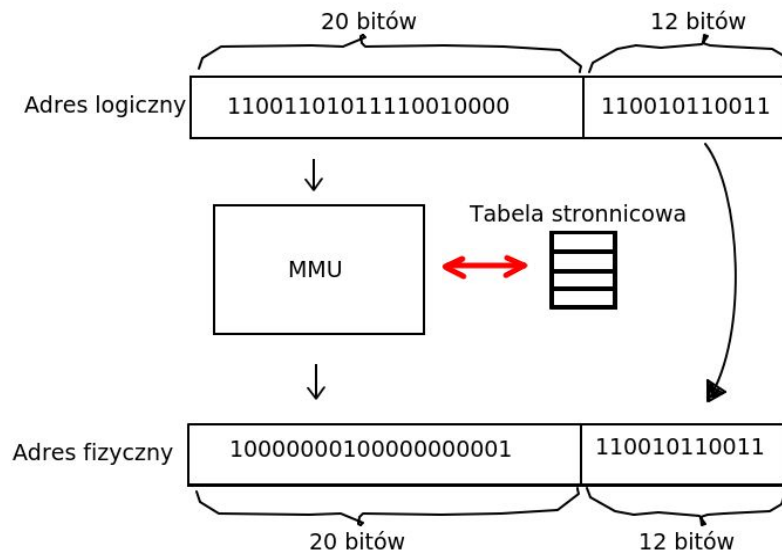
2. Zarządzanie pamięcią.

- **b) Stronicowanie i tabele stronicowe**
- **Jak odbywa się adresowanie ?**
- .
- 12 bitów przesunięcia jest kopiowanych bezpośrednio z adresu logicznego na adres fizyczny.



2. Zarządzanie pamięcią.

- **b) Stronicowanie i tabele stronicowe**
- **Jak odbywa się adresowanie ?**
 -
 - 12 bitów przesunięcia jest kopiowanych bezpośrednio z adresu logicznego na adres fizyczny.
 - Natomiast 20 bitów adresu logicznego służy do znalezienia w tabeli stronicowej odpowiedniego adresu fizycznego.

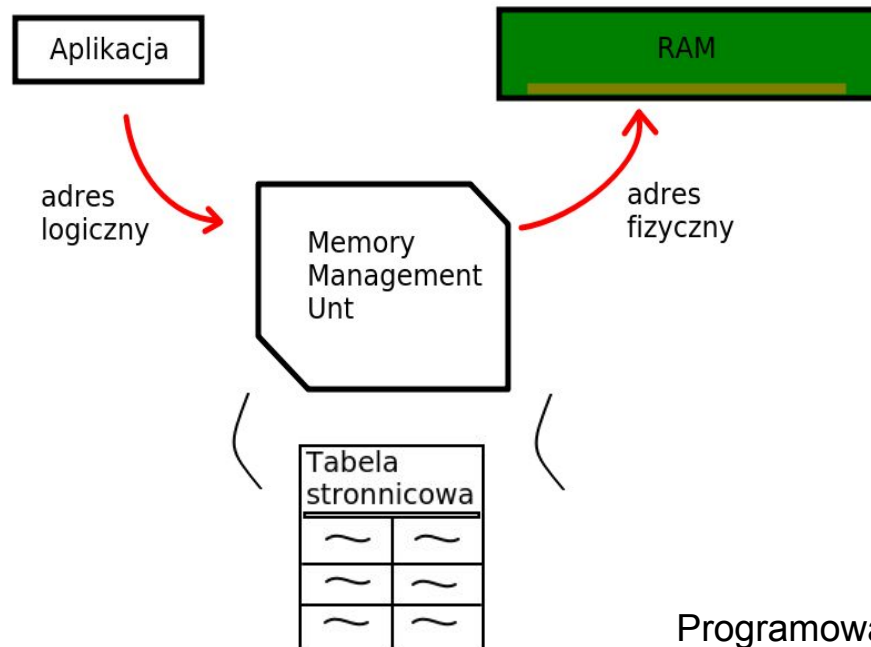


2. Zarządzanie pamięcią.

- b) Stronicowanie i tabele stronicowe

- **Problem:**

- .
- Tłumaczenie każdej strony wymaga spojrzenia na **tabelę stronicową**, która jest umieszczona w pamięci RAM.
- Niewydajne, dwukrotny dostęp do pamięci przy pojedynczym odwołaniu danych



Programowanie
niskopoziomowe

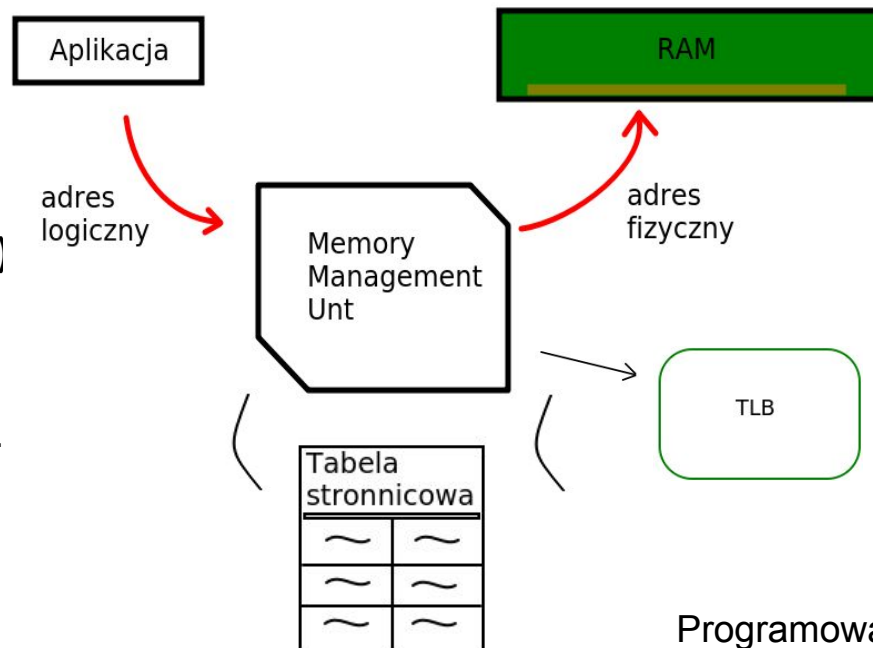
2. Zarządzanie pamięcią.

b) Stronicowanie I tabele stronicowe

Rozwiązanie:

• Bufor mikroprocesorowej pamięci (Translation Lookaside Buffer – TLB)

- Jest to pamięć podręczna, która przechowuje część mapowań adresów.
- Szybka, wydajna, lecz droga
- Typowy rozmiar – 8 do 4096 wejść.



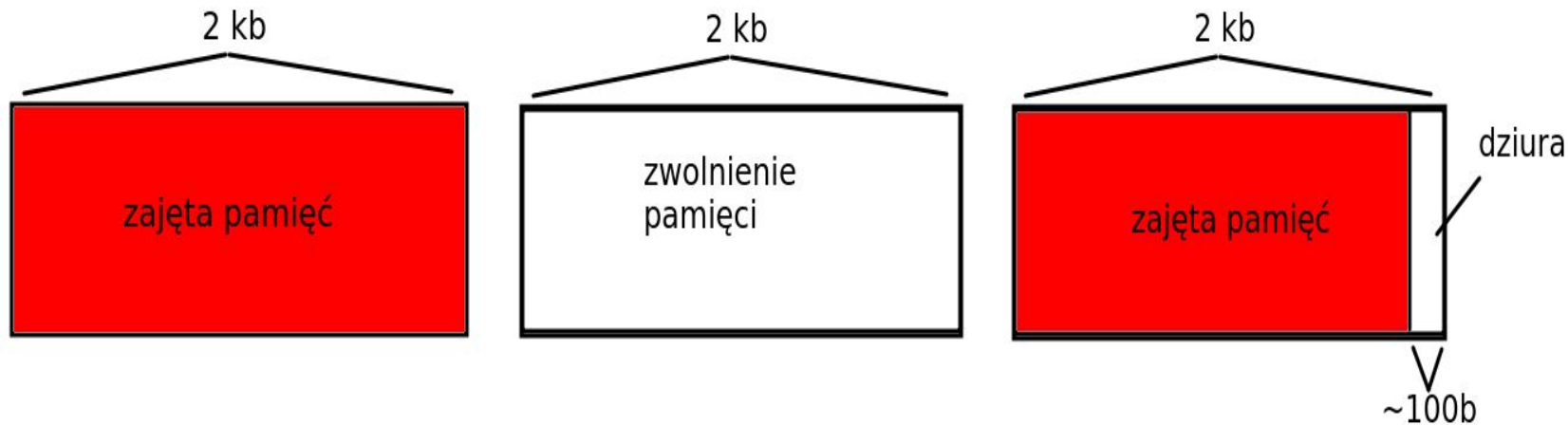
Programowanie
niskopoziomowe

2. Zarządzanie pamięcią.

- b) Stronicowanie i tabele stronicowe
- **Zalety stronicowania:**
- Główną zaletą stronicowania jest **brak fragmentacji zewnętrznej**, czyli sytuacji gdy w pamięci powstają dziury – z których już nikt nie korzysta.
- **Przykład ---->**

2. Zarządzanie pamięcią.

b) Stronicowanie I tabele stronicowe



2. Zarządzanie pamięcią.

- b) Stronicowanie i tabele stronicowe
- **Zalety stronicowania:**
- Główną zaletą stronicowania jest **brak fragmentacji zewnętrznej**.
- Przydzielanie pamięci w kawałkach ustalonego rozmiaru zapobiega fragmentacji zewnętrznej gdyż najmniejszy nieużyty blok pamięci ma rozmiar jednej **strony**.
- Proces, który zgłasza zapotrzebowanie na kilka ramek dostaje je pojedynczo, co powoduje, że nie muszą być one spójnym blokiem pamięci fizycznej.

2. Zarządzanie pamięcią.

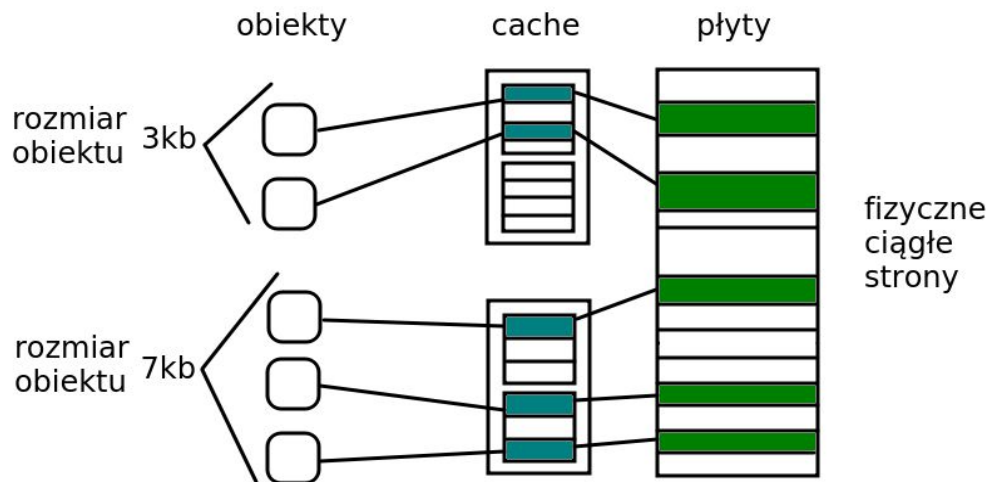
- b) Stronicowanie i tabele stronicowe
- **Wady stronicowania:**
- **Fragmentacja wewnętrzna** – proces może potrzebować mniej pamięci niż 4 kb pochodzące od **strony** – pozostała część strony może być nie użyta.
- Koszt utrzymania pamięci w postaci stron i ramek

2. Zarządzanie pamięcią.

- c) slab allocator & buddy system
- **slab allocator (alokator płytowy)** oraz **buddy system (system bliźniaków)**
tę dwa mechanizmy radzenia sobie z alokacją pamięci

2. Zarządzanie pamięcią.

- c) slab allocator
- **Alokator płytowy** – mechanizm zarządzania pamięcią przeznaczony do wydajnej alokacji pamięci obiektów



2. Zarządzanie pamięcią.

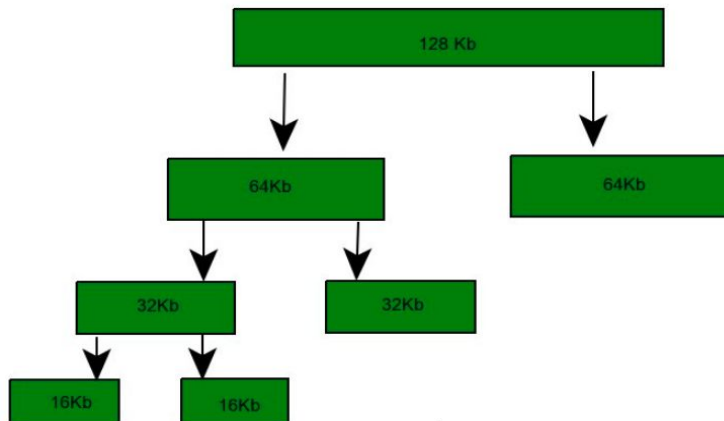
- **c) slab allocator**
 - - wykorzystuje mechanizm cachowania (pamięci podręcznej)
 - - osobny cache na obiekty plików, struktury danych reprezentujące deskryptory procesów, semaforey
 - - w tej technice kawałki pamięci (**memory chunks**), które mogłyby pomieścić pewne obiekty są poprzednio prealokowane

2. Zarządzanie pamięcią.

- c) slab allocator
- **ZALETY:**
 - - pełne wykorzystanie pamięci, brak fragmentacji
 - - żądanie o obszar pamięci może być obsłużone bardzo sprawnie
 - - dobre dla obiektów mniejszych niż strona
- **WADY:**
 - - narzut pamięci ze względu na cache

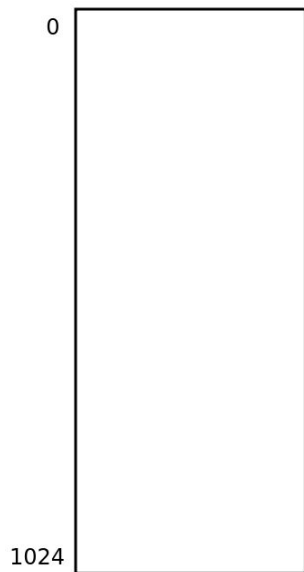
2. Zarządzanie pamięcią.

- c) buddy system
- **Buddy system (system bliźniaków)** - metoda alokacji pamięci, która charakteryzuje się dużą szybkością i łatwością implementacji oraz niską fragmentacją zewnętrzną, kosztem jednak znaczącej fragmentacji wewnętrznej.



2. Zarządzanie pamięcią.

- c) buddy system



A - żąda alokacji 32 mb

B - żąda alokacji 64 mb

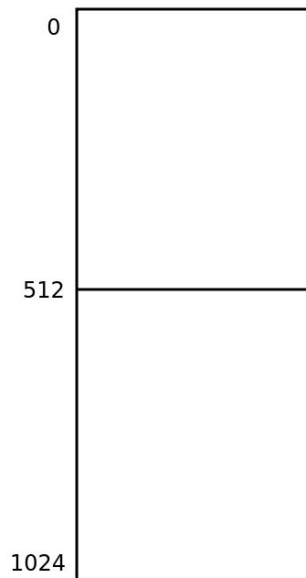
C - żąda alokacji 60 mb

D - żąda alokacji 150mb

procesy

2. Zarządzanie pamięcią.

- c) buddy system



A - żąda alokacji 32 mb

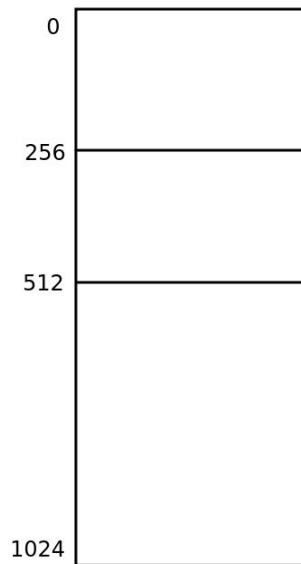
B - żąda alokacji 64 mb

C - żąda alokacji 60 mb

D - żąda alokacji 150mb

2. Zarządzanie pamięcią.

- c) buddy system



A - żąda alokacji 32 mb

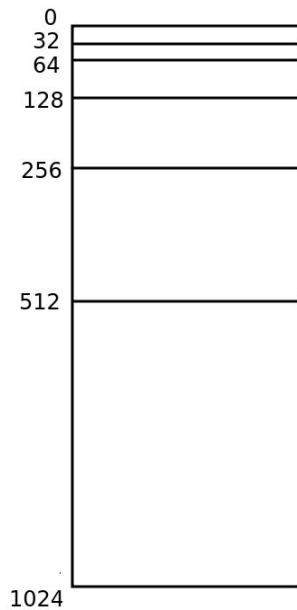
B - żąda alokacji 64 mb

C - żąda alokacji 60 mb

D - żąda alokacji 150mb

2. Zarządzanie pamięcią.

c) buddy system



A - żąda alokacji 32 mb

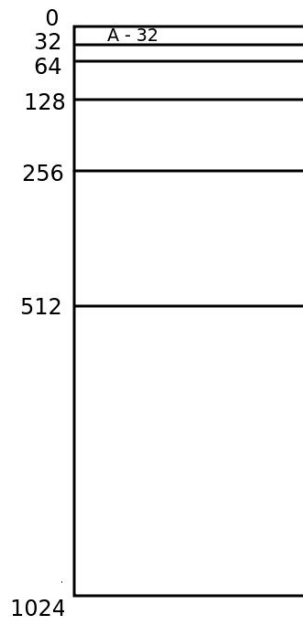
B - żąda alokacji 64 mb

C - żąda alokacji 60 mb

D - żąda alokacji 150mb

2. Zarządzanie pamięcią.

c) buddy system



~~A - żąda alokacji 32 mb~~

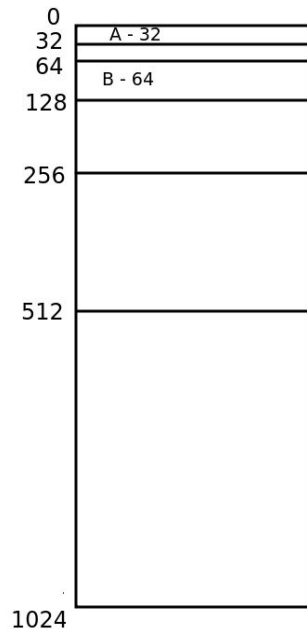
B - żąda alokacji 64 mb

C - żąda alokacji 60 mb

D - żąda alokacji 150mb

2. Zarządzanie pamięcią.

c) buddy system



~~A - żąda alokacji 32 mb~~

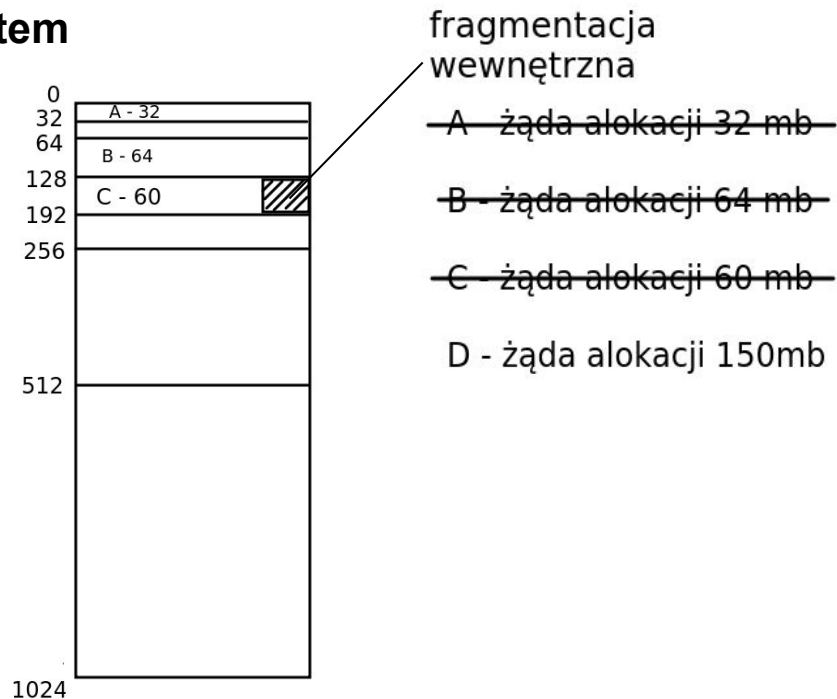
~~B - żąda alokacji 64 mb~~

C - żąda alokacji 60 mb

D - żąda alokacji 150mb

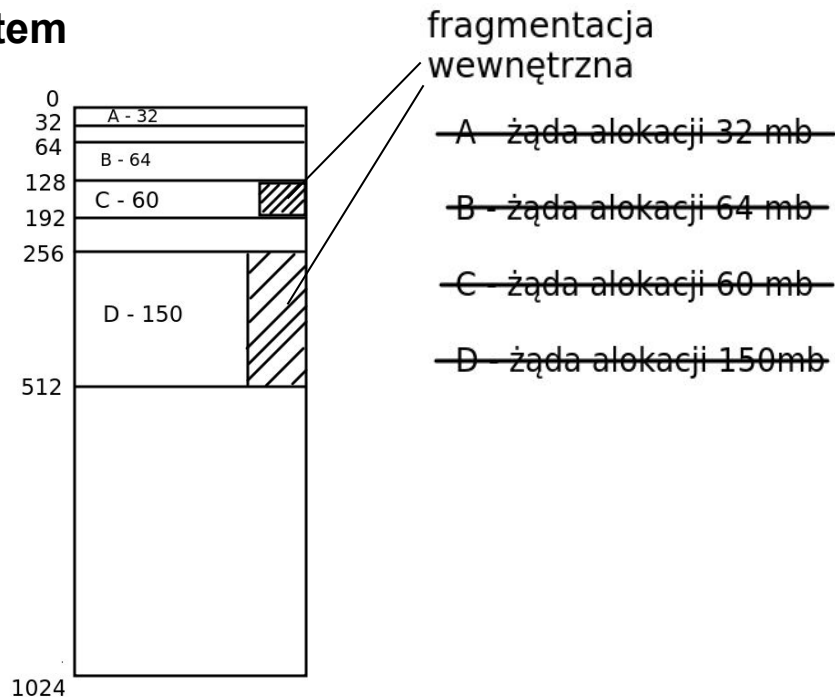
2. Zarządzanie pamięcią.

c) buddy system



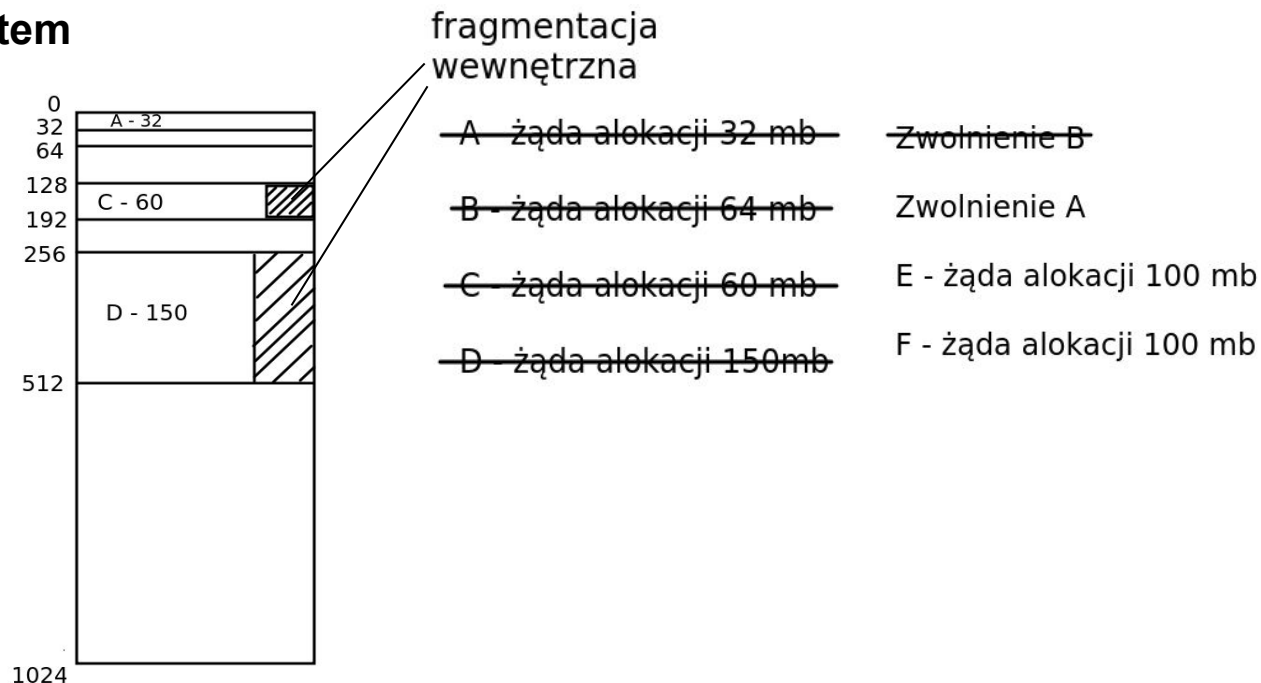
2. Zarządzanie pamięcią.

c) buddy system



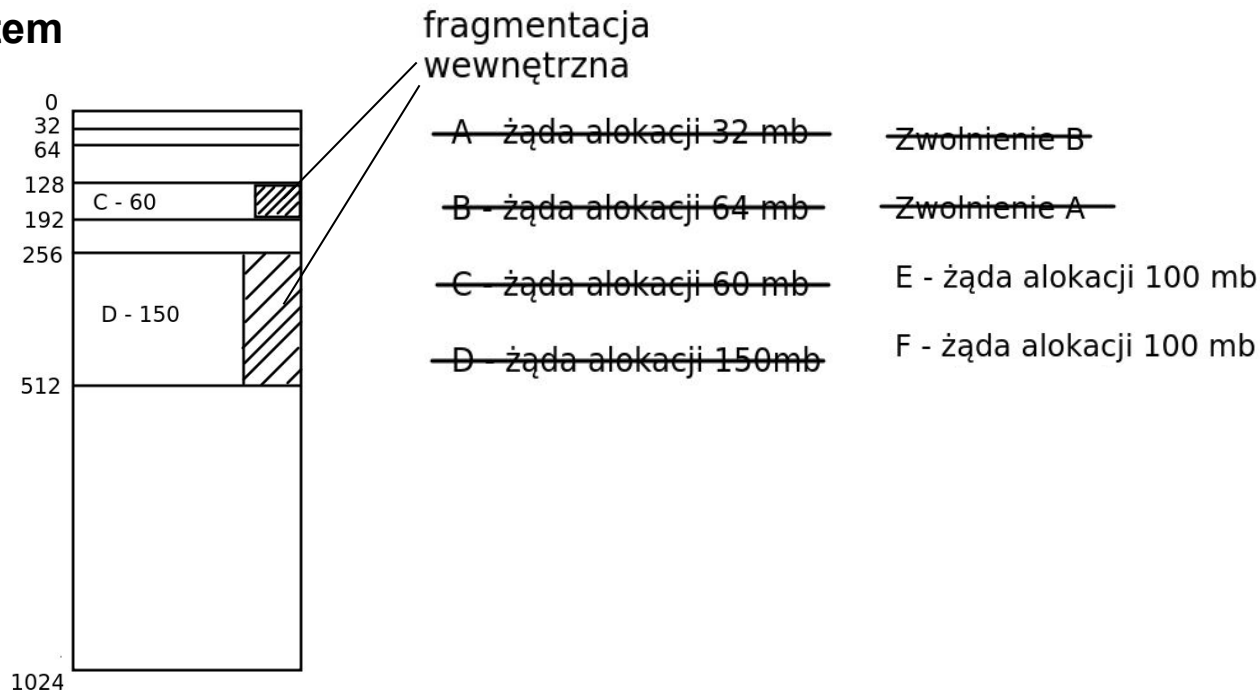
2. Zarządzanie pamięcią.

c) buddy system



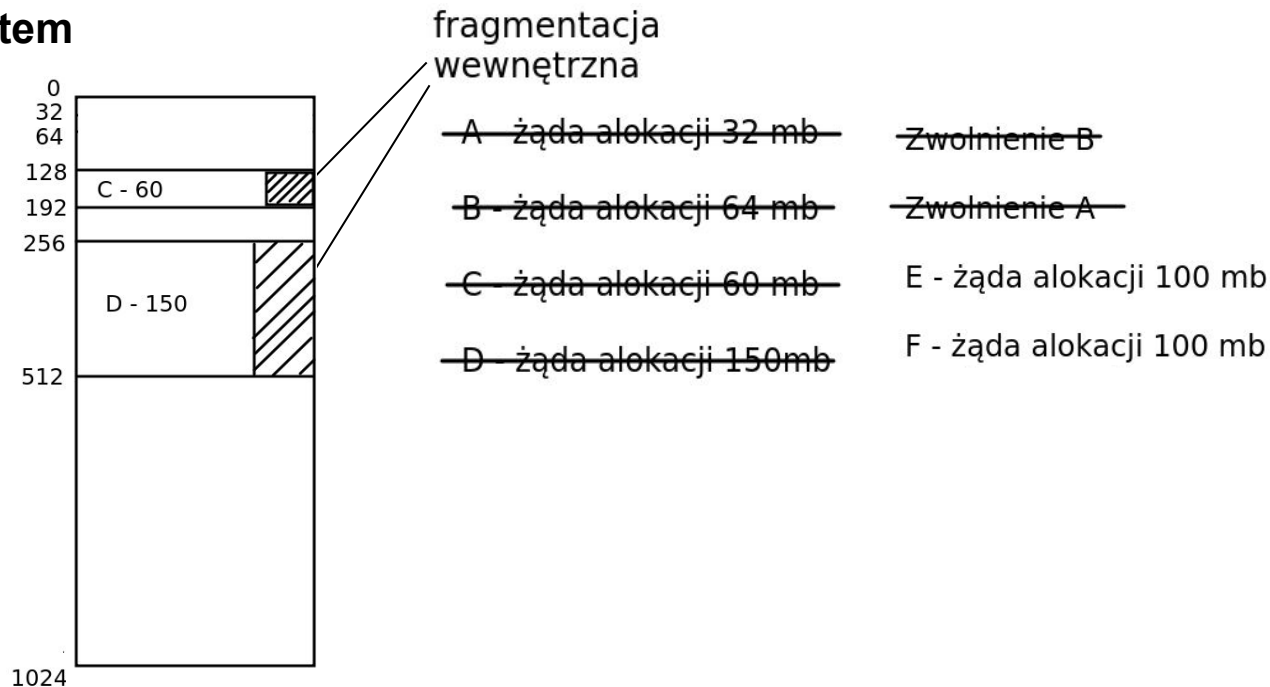
2. Zarządzanie pamięcią.

c) buddy system



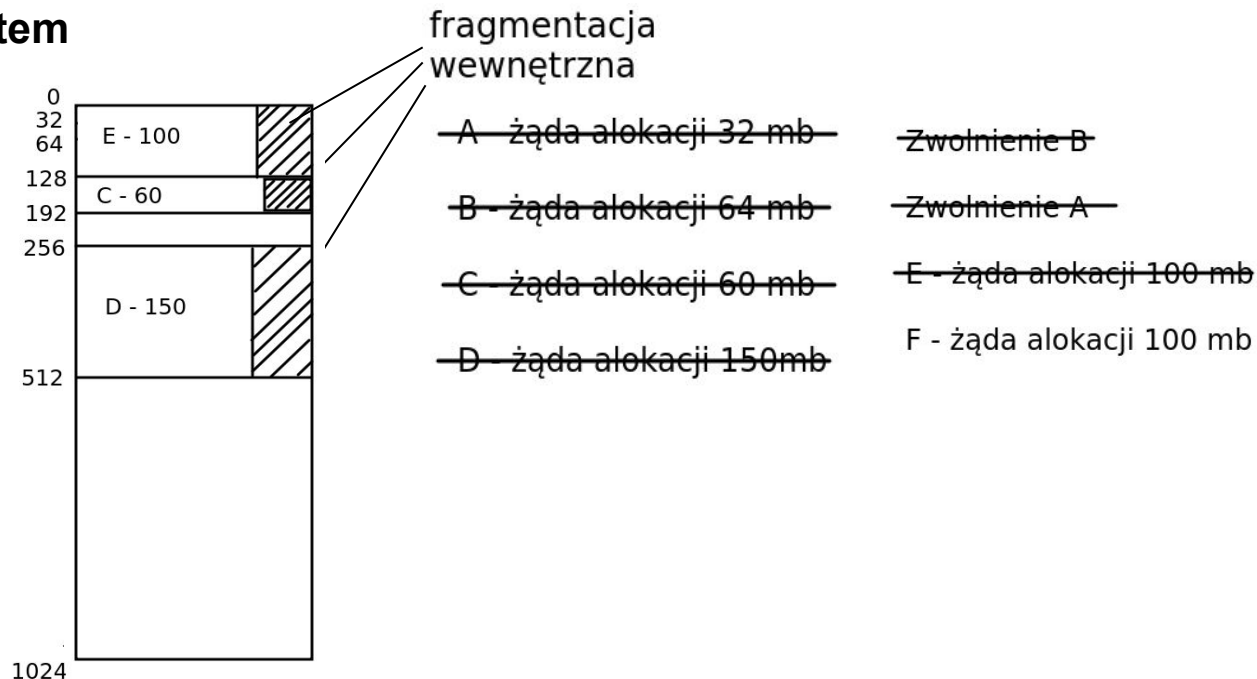
2. Zarządzanie pamięcią.

c) buddy system



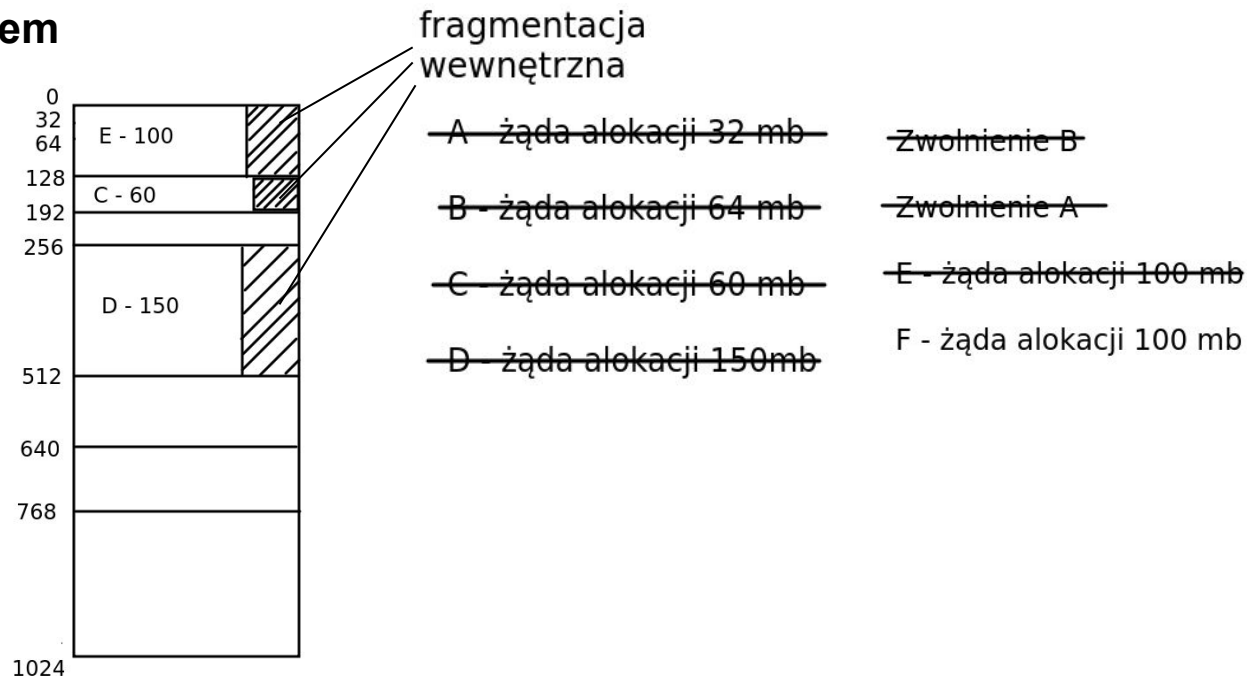
2. Zarządzanie pamięcią.

c) buddy system



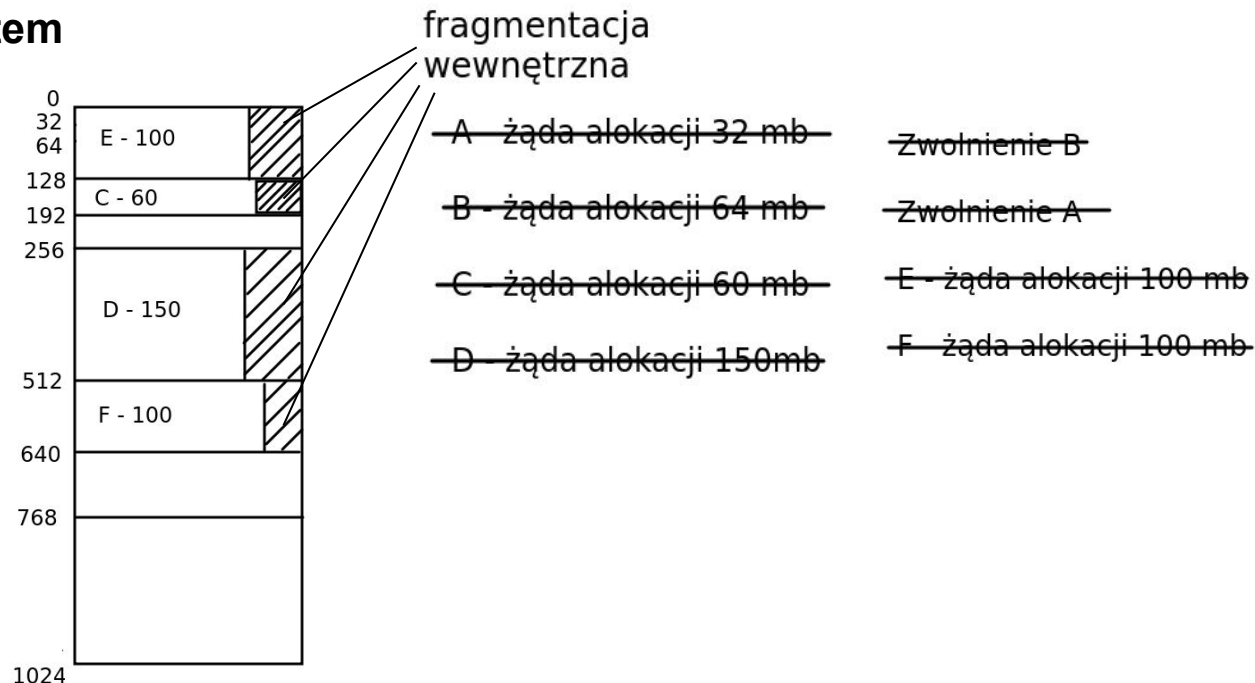
2. Zarządzanie pamięcią.

c) buddy system



2. Zarządzanie pamięcią.

c) buddy system



2. Zarządzanie pamięcią.

- **c) buddy system**
- **ZALETY:**
 - - mała fragmentacja zewnętrzna
 - - szybki, ze względu na strukturę drzewa binarnego
 - - łączenie w większe bloki
 - - prosta kalkulacja adresów
- **WADY:**
 - - występuje fragmentacja wewnętrzna (która w konsekwencji może powodować znaczne straty w postaci nieużytej pamięci!)

Zarządzanie procesami

- a) priorytety i przydział procesora
- b) typy procesów (Kernel process/user process)
- c) przekazywanie danych pomiędzy procesami
- d) planer
- e) startowanie, kopiowanie, usuwanie procesów

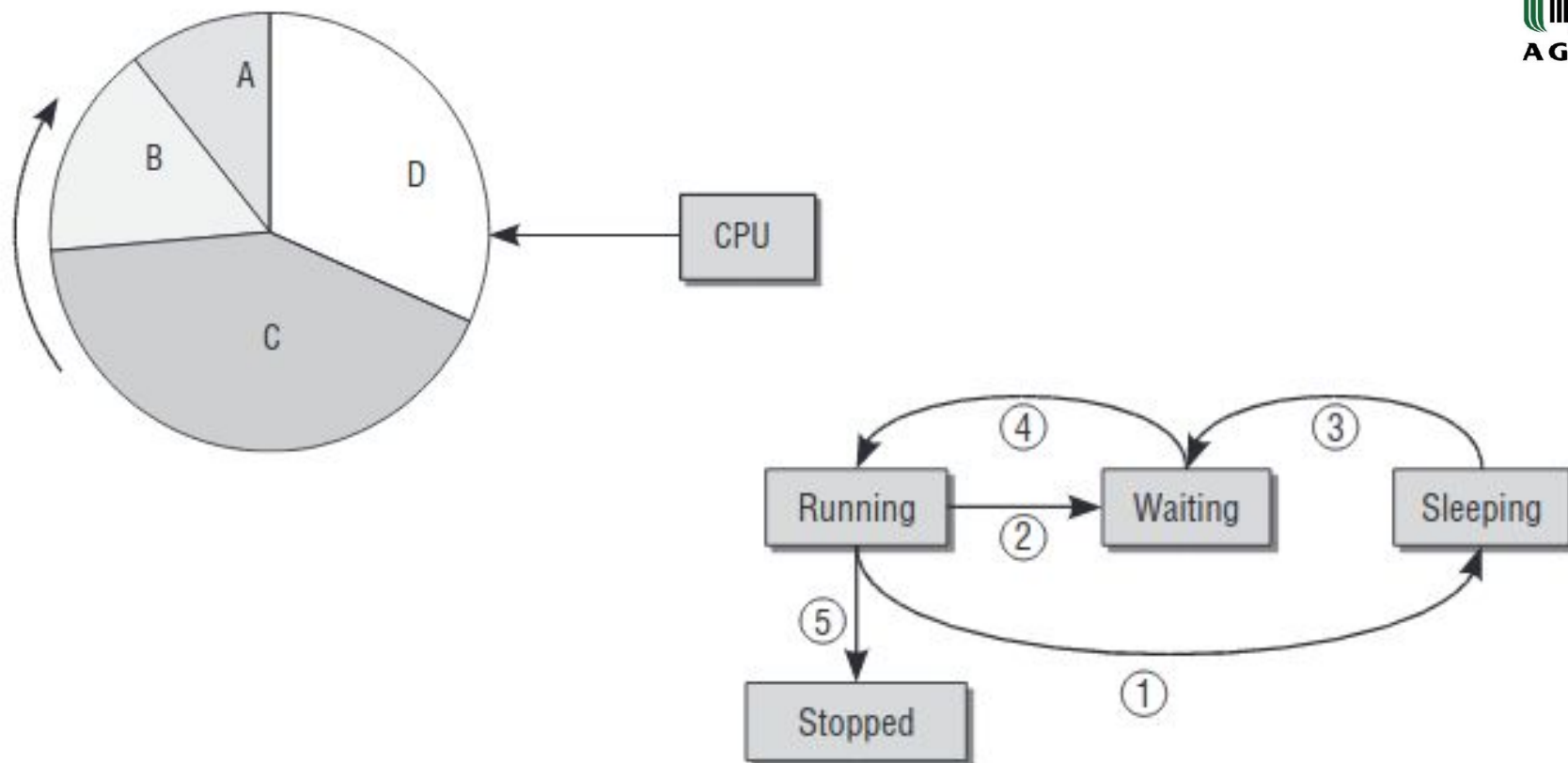
Identyfikacja procesów

PID - Process Identification Number

TGID - Thread Group ID

PGID - Process Group ID

SID - Session ID



Klasyfikacja procesów ze względu na potrzeby czasowe

- Hard real-time process - Procesy z potrzebą wykonania w przeciągu danego czasu, zazwyczaj jak najszybciej. (nieobsługiwane przez kernel linuxa)
- Soft real-time process - Proces z potrzebą wykonania w przeciągu danego czasu, ale dopuszczalne są opóźnienia.
- Normal process - Zwyczajny proces użytkownika, w naszych systemach korzystamy głównie z takich, wykonuje się kiedy zostanie przydzielony CPU.

Priorytety

```
ps -e -o uid,pid,ppid,pri,ni,cmd
```

Tryby jądra(wykonywania procesów)

- Zwykły - tryb w którym wykonywane są zwykłe procesy, mogą być wywłaszczone przez inne procesy, system call i przerwania.
- Kernel - tryb w którym przetwarzane są system call. Może być wywłaszczony jedynie przez przerwanie.

Komunikacja międzyprocesowa(IPC)

Używamy gdy:

- Dwa(lub więcej) procesy dzielą dane
- Proces A czeka na wykonanie procesu B
- Proces A przekazuje dane procesowi B

Metody komunikacji pochodzą z System V

Semafor

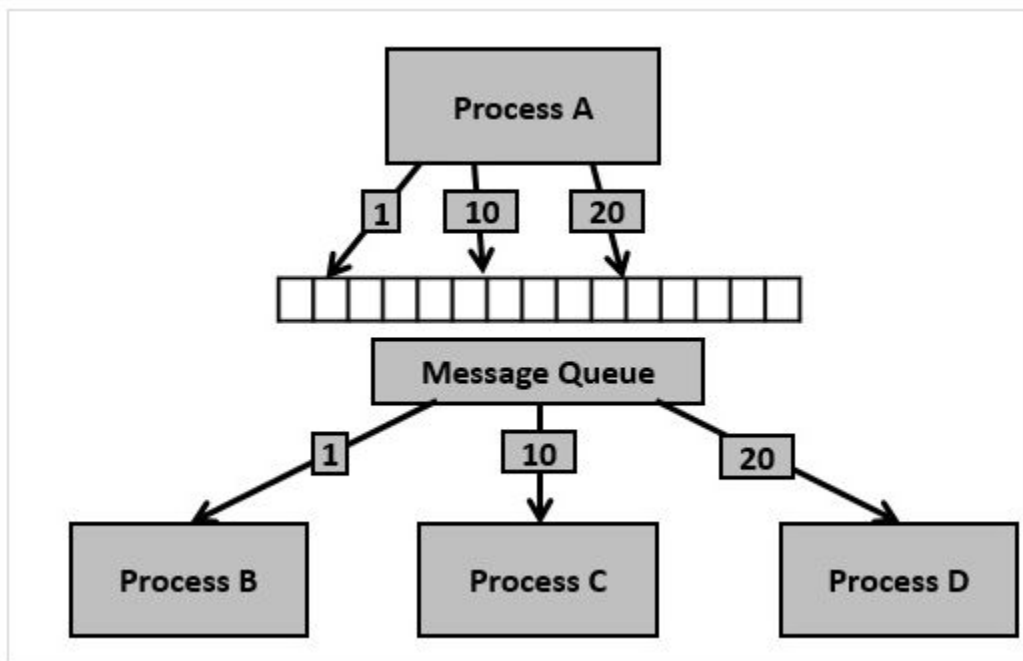
Najprostsza metoda uzgadniania dostępu do danych. Kilka procesów dzieli wspólną zmienną. Arbitralnie ustawiamy klucz semafora(wspólny dla wszystkich procesów)

Dostęp do współdzielonych danych:

1. Ustawienie semafora o podanym kluczu
2. Przy dostępie do danych obniża wartość semafora o 1 i uzyskuje dostęp do danych, lub jeśli semafor już posiada wartość 0, proces “zasypia” i czeka, aż będzie mógł uzyskać dostęp do danych.
3. Kiedy kończy korzystać z danych, zwiększa wartość semafora o 1 i wybudza pierwszy proces w kolejce.

[Przykład](#)

Kolejki wiadomości



Wysłanie wiadomości kolejką:

1. Wygenerowanie klucza kolejki
2. Stworzenie lub pobranie numeru kolejki z pomocą klucza z punktu 1.
3. Ustawienia w strukturze numeru wiadomości i wiadomości
4. Wysłanie wiadomości

Odebranie wiadomości z kolejki:

1. Punktu 1,2 z wysyłki.
2. Przeszukanie kolejki w ramach znalezienia wiadomości o wybranym numerze.

UWAGA: Wiadomość dotrze jedynie do pierwszego procesu który ją odbierze z kolejki!

[Przykład](#)

Sygnaly

Komunikacja za pomocą sygnałów następuje za pomocą wysyłania sygnałów i ich obsługę przez handlery.

[Przykład](#)

Planer

Program zarządzający przełączaniem procesów wykonywanych na CPU. Jego głównym zadaniem jest sprawiedliwe rozdzielenie zasobów pomiędzy procesami z uwzględnieniem ich priorytetów. Zawsze najpierw rozpatrujemy procesy real-time, potem zaczynamy rozpatrywać procesy normalne.

Planer w linuxie obecnie korzysta z dwóch polityk planowania: Real-Time Scheduler i Completely Fair Scheduler.

Problemy które musi rozwiązywać planer

- Unikanie zbyt częstego przełączania procesów
- Unikanie zbyt rzadkiego przełączania procesów
- “Równy” podział czasu procesora
- Obsługa wielu rdzeni

Działanie planera w linuxie

- Wszystkie procesy są przetrzymywane w drzewie czerwono-czarnym, używając przydzielonego czasu procesora jako index.
- Każdy proces ma przypisany maksymalny czas wykonania
- Gdy planer musi wywołać proces to wywołuje proces z najmniejszym indexem
- W zależności od priorytetu procesu różnie naliczany mu jest czas procesora. Np. proces z priorytetem 5 za każdą nanosekundę ma naliczane 3 nanosekundy.

Tworzenie nowych procesów

Fork - “kopiowanie” procesu, kopiujemy całą instancję procesu, razem z pamięcią, co jest wolne i zasobożerne. Aby tego uniknąć linux używa techniki “Copy On Write”.

“Copy On Write” zamiast kopiować całe strony pamięci udostępnia procesów “read-only” dostęp do danych pierwotnego procesu. W momencie próby nadpisania danych przez którykolwiek z procesów proces ten “kopiuje” nadpisane dane i zmienia ich adres w tablicy pamięci stronicowej.

[Dokumentacja](#)

Tworzenie nowych procesów

vfork - skopiowanie procesu, ale z dostępem do tych samych danych co oryginalny proces.

[Dokumentacja](#)

Tworzenie nowych procesów

Clone - Klonowanie z wyborem które dokładnie dane chcemy skopiować.

[Dokumentacja](#)

Tworzenie nowych procesów

Exec - Zastępuje wywołujący go proces innym procesem podanym jako argument.

[Dokumentacja](#)

“Zabijanie” procesów

kill(PID, SIGTERM / SIGKILL)
/usr/include/signal.h

Reprezentacja procesu w C

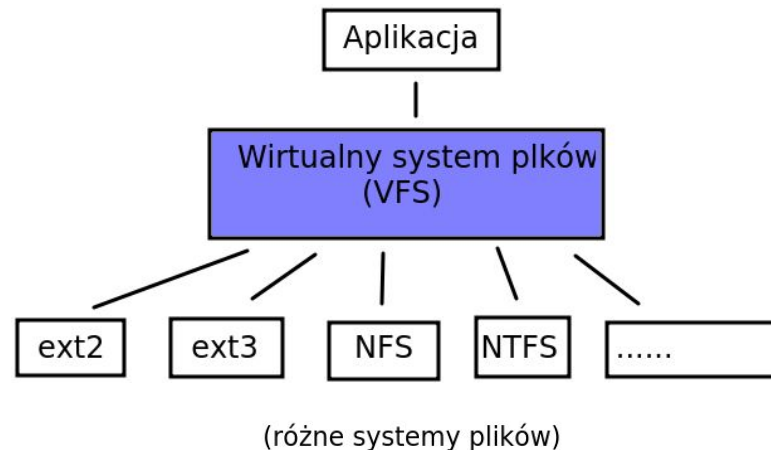
[usr/include/sched.h](#)

4. Wirtualny system plików (VFS).

- » a) typy systemów plików
- » b) powszechny model plików (Common File Model)
- » c) struktura Virtual File System(VFS)

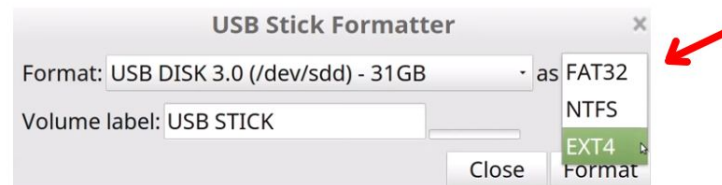
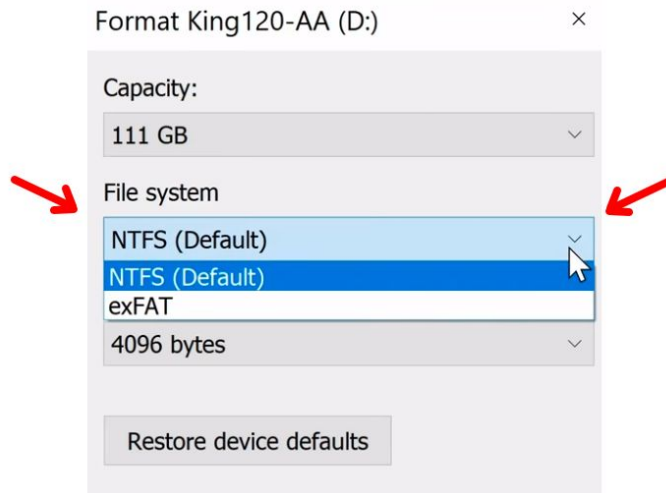
4. Wirtualny system plików (VFS).

- » **Wirtualny system plików**
- »
 - jest to abstrakcyjna warstwa pośrednicząca między właściwym systemem plików a jądrem systemu operacyjnego.
 - w pewnym sensie jest to kontener dostarczający funkcjonalność systemu plików.



4. Wirtualny system plików (VFS).

» a) typy systemów plików



4. Wirtualny system plików (VFS).

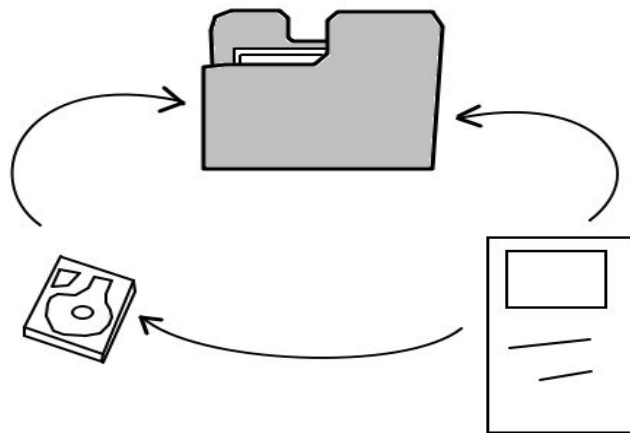
- » a) typy systemów plików
- » **Second Extended File System (ext2)** – wprowadzony w 1993 roku system plików dla systemu operacyjnego Linux.
 - maksymalny rozmiar pliku: od **16 GB** do **2 TB**
- »
 - maksymalny rozmiar systemu: od **2 TB** do **32 TB**
- »
 - brak funkcji kronikowania
(księgowania)

4. Wirtualny system plików (VFS).

- » a) typy systemów plików
- » **Third Extended File System (ext3)** – wprowadzony w 2001 roku system plików dla systemu operacyjnego Linux.
 - maksymalny rozmiar pliku: od **16 GB** do **2 TB**
 - »
 - maksymalny rozmiar systemu: od **2 TB** do **32 TB**
 - »
 - wprowadzona **funkcja kronikowania (księgowania)**

4. Wirtualny system plików (VFS).

- » a) typy systemów plików
- » Czym jest funkcja kronikowania (księgowania)?



4. Wirtualny system plików (VFS).

- » a) typy systemów plików
- » **Reiser File System (RFS)** – uniwersalny, kronikowalny system plików stworzony przez zespół Namesys (2001 rok).
- »
 - maksymalny rozmiar pliku: **8 TB**
 - maksymalny rozmiar systemu: **16 TB**
- » - posiada **funkcję kronikowania (księgowania)**
- » - szybszy od **ext2** i **ext3**

4. Wirtualny system plików (VFS).

- » a) typy systemów plików
- » **Fourth Extended File System (ext4)** – następca **ext3**, wprowadzony w 2008 roku
 - maksymalny rozmiar pliku: od **16 GB** do **16 TB**
- »
 - maksymalny rozmiar systemu: **1 EB** (eksabajt, **1 EB = 10⁹ GB**) !!!
- » - posiada **funkcję kronikowania (księgowania)**
- »

4. Wirtualny system plików (VFS).

» b) powszechny model plików (Common File Model)

» **“Wszystko jest plikiem”**

»

4. Wirtualny system plików (VFS).

- » b) powszechny model plików (Common File Model)

```
[root@ip-10-0-0-105 ~]# ls -l
total 4
-rw-r--r--. 1 root root 12 Apr  8 02:03 basictextfile.txt
drwxr-xr-x. 2 root root  6 Apr  8 02:03 directory
[root@ip-10-0-0-105 ~]#
```

»

4. Wirtualny system plików (VFS).

- » b) powszechny model plików (Common File Model)

```
[root@ip-10-0-0-105 ~]# ls -l
total 4
-rw-r--r--. 1 root root 12 Apr  8 02:03 basictextfile.txt
drwxr-xr-x. 2 root root  6 Apr  8 02:03 directory
[root@ip-10-0-0-105 ~]#
```

»

Katalog jest plikiem.

4. Wirtualny system plików (VFS).

» b) powszechny model plików (Common File Model)

»

```
[root@ip-10-0-0-105 bin]# ls -l *ls*
-rwxr-xr-x. 1 root root 28896 Feb 16 2016 false
-rwxr-xr-x. 1 root root 243392 Jan 5 2016 grub2-menulst2cfg
-rwxr-xr-x. 1 root root 117616 Feb 16 2016 ls
-rwxr-xr-x. 1 root root 11520 Mar 5 2015 lsattr
-rwxr-xr-x. 1 root root 72592 Nov 20 2015 lsblk
-rwxr-xr-x. 1 root root 58352 Nov 20 2015 lscpu
-rwxr-xr-x. 1 root root 6632 Dec 3 2015 lsinitrd
-rwxr-xr-x. 1 root root 37648 Nov 20 2015 lslocks
-rwxr-xr-x. 1 root root 88136 Nov 20 2015 lslogins
-rwxr-xr-x. 1 root root 2614 Sep 10 2014 lss3
-rwxr-xr-x. 1 root root 292192 Feb 16 2016 systemd-cgls
```

4. Wirtualny system plików (VFS).

» b) powszechny model plików (Common File Model)

```
[root@ip-10-0-0-105 bin]# ls -l *ls*
-rwxr-xr-x. 1 root root 28896 Feb 16 2016 false
-rwxr-xr-x. 1 root root 243392 Jan 5 2016 grub2-menulst2cfg
-rwxr-xr-x. 1 root root 117616 Feb 16 2016 ls
-rwxr-xr-x. 1 root root 11520 Mar 5 2015 lsattr
-rwxr-xr-x. 1 root root 72592 Nov 20 2015 lsblk
-rwxr-xr-x. 1 root root 58352 Nov 20 2015 lscpu
-rwxr-xr-x. 1 root root 6632 Dec 3 2015 lsinitrd
-rwxr-xr-x. 1 root root 37648 Nov 20 2015 lslocks
-rwxr-xr-x. 1 root root 88136 Nov 20 2015 lslogins
-rwxr-xr-x. 1 root root 2614 Sep 10 2014 lss3
-rwxr-xr-x. 1 root root 292192 Feb 16 2016 systemd-cgls
```

Plik wykonywalny także
jest zwykłym plikiem.

4. Wirtualny system plików (VFS).

» b) powszechny model plików (Common File Model)

»

```
[root@ip-10-0-0-105 dev]# ls
1          filetoprint.txt  net          shm          tty15      tty28      tty40      tty53      tty9       vcsa6
autofs     full                     network_latency  snapshot   tty16      tty29      tty41      tty54      ttyS0      vfblo
block      fuse                     network_throughput  snd        tty17      tty3       tty42      tty55      ttyS1      vga_arbiter
btrfs-control  hpet                     null          stderr      tty18      tty30      tty43      tty56      ttyS2      vhost-net
char       hugepages               nvram         stdin        tty19      tty31      tty44      tty57      ttyS3      xen
console    initctl                 oldmem        stdout       tty2       tty32      tty45      tty58      uhid       xvda
core       input                   port          tty          tty20      tty33      tty46      tty59      uinput     xvda1
cpu        kmsg                    ppp           tty0         tty21      tty34      tty47      tty6       urandom     zero
cpu_dma_latency  log                     ptmx          tty1         tty22      tty35      tty48      tty60      usbmon0
crash      loop-control            pts           tty10        tty23      tty36      tty49      tty61      vcs
disk       mapper                  random        tty11        tty24      tty37      tty5       tty62      vcs1
dri        mcelog                  raw           tty12        tty25      tty38      tty50      tty63      vcs6
fb0        mem                     rtc           tty13        tty26      tty39      tty51      tty7       vcsa
fd         mqueue                  rtc0          tty14        tty27      tty4       tty52      tty8       vcsa1

[root@ip-10-0-0-105 dev]# ls -l xvda
brw-rw----. 1 root disk 202, 0 Apr  2 23:29 xvda
```

4. Wirtualny system plików (VFS).

» b) powszechny model plików (Common File Model)

»

```
[root@ip-10-0-0-105 dev]# ls
1          filetoprint.txt  net          shm          tty15      tty28      tty40      tty53      tty9       vcsa6
autofs     full                     network_latency  snapshot   tty16      tty29      tty41      tty54      ttyS0      vflo
block      fuse                     network_throughput  snd        tty17      tty3       tty42      tty55      ttyS1      vga_arbiter
btrfs-control  hpet                     null          stderr      tty18      tty30      tty43      tty56      ttyS2      vhost-net
char       hugepages                nvram         stdin        tty19      tty31      tty44      tty57      ttyS3      xen
console    initctl                  oldmem        stdout       tty2       tty32      tty45      tty58      uhid       xvda
core       input                    port          tty          tty20      tty33      tty46      tty59      uinput     xvda1
cpu        kmsg                     ppp           tty0         tty21      tty34      tty47      tty6       urandom    zero
cpu_dma_latency  log                     ptmx          tty1         tty22      tty35      tty48      tty60      usbmon0
crash      loop-control             pts           tty10        tty23      tty36      tty49      tty61      vcs
disk       mapper                   random        tty11        tty24      tty37      tty5       tty62      vcs1
dri        mcelog                   raw           tty12        tty25      tty38      tty50      tty63      vcs6
fb0        mem                      rtc           tty13        tty26      tty39      tty51      tty7       vcsa
fd         mqueue                  rtc0          tty14        tty27      tty4       tty52      tty8       vcsa1

[root@ip-10-0-0-105 dev]# ls -l xvda
brw-rw----. 1 root disk 202, 0 Apr  2 23:29 xvda
```

Urządzenia również
są plikami.

4. Wirtualny system plików (VFS).

- » **c) struktura VFS**
- » - Wirtualny System Plików jest zorientowany obiektowo.
- » - Rodzina struktur danych reprezentują powszechny model pliku.
- » - Struktury zawierają dane oraz wskaźniki do funkcji systemów plików.
- »

4. Wirtualny system plików (VFS).

- » **c) struktura VFS**
- » Wyróżnia się cztery główne obiekty VFS:
 - » **Superblock object**
 - » **Inode object**
 - » **Dentry object**
 - » **File object**
- »

4. Wirtualny system plików (VFS).

- » **c) struktura VFS**
- » **Superblock object**
- » - reprezentuje zamontowany system plików
- » - operacje na superblocku zawarte w strukturze **super_operations**

- »

4. Wirtualny system plików (VFS).

- » c) struktura VFS
- » Superblock object

»

»

```

struct super_block {
    struct list_head    s_list;        /* list of all superblocks */
    dev_t               s_dev;         /* identifier */
    unsigned long        s_blocksize;  /* block size in bytes */
    unsigned char        s_blocksize_bits; /* block size in bits */
    unsigned char        s_dirt;       /* dirty flag */
    unsigned long        s_maxbytes;   /* max file size */
    struct file_system_type s_type;    /* filesystem type */
    struct super_operations s_op;      /* superblock methods */
    struct dquot_operations *dq_op;    /* quota methods */
    struct quotactl_ops   *s_qcops;    /* quota control methods */
    struct export_operations *s_export_op; /* export methods */
    unsigned long         s_flags;     /* mount flags */
    unsigned long         s_magic;     /* filesystem's magic number */
    struct dentry         *s_root;     /* directory mount point */
    struct rw_semaphore   s_umount;    /* unmount semaphore */
    struct semaphore      s_lock;      /* superblock semaphore */
    int                   s_count;     /* superblock ref count */
    int                   s_need_sync; /* not-yet-synced flag */
    atomic_t              s_active;    /* active reference count */
    void                  *s_security; /* security module */
    struct xattr_handler **s_xattr;    /* extended attribute handlers */
    struct list_head      s_inodes;    /* list of inodes */
    struct list_head      s_dirty;     /* list of dirty inodes */
    struct list_head      s_io;       /* list of writebacks */
    struct list_head      s_more_io;  /* list of more writeback */
    struct hlist_head     s_anon;     /* anonymous dentries */
    struct list_head      s_files;    /* list of assigned files */
    struct list_head      s_dentry_lru; /* list of unused dentries */
    int                   s_nr_dentry_unused; /* number of dentries on list */
    struct block_device   *s_bdev;    /* associated block device */
    struct mtd_info        *s_mtd;     /* memory disk information */
    struct list_head      s_instances; /* instances of this fs */
    struct quota_info      s_dquot;    /* quota-specific options */
    int                   s_frozen;    /* frozen status */
    wait_queue_head_t     s_wait_unfrozen; /* wait queue on freeze */
    char                  s_id[32];    /* text name */
    void                  *s_fs_info;  /* filesystem-specific info */
    fmode_t               s_mode;     /* mount permissions */
    struct semaphore      s_vfs_rename_sem; /* rename semaphore */
    u32                   s_time_gran; /* granularity of timestamps */
    char                  *s_subtype; /* subtype name */
    char                  *s_options; /* saved mount options */
};
  
```

»

4. Wirtualny system plików (VFS).

- » **c) struktura VFS**
- » **Superblock object**
- » - operacje na
superblocku zawarte
w strukturze
super_operations
- »
- »

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, int);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    int (*show_options)(struct seq_file *, struct vfsmount *);
    int (*show_stats)(struct seq_file *, struct vfsmount *);
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
    int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

4. Wirtualny system plików (VFS).

- » c) struktura VFS
- » Inode object
- »
 - reprezentuje wszystkie informacje potrzebne jądro systemu do manipulacji plikami i katalogami.
- » - operacje na **inode** zawarte w strukturze **inode_operations**
- »

4. Wirtualny system plików (VFS).

» c) struktura VFS

» Inode object

»

»

»

```

struct inode {
    struct hlist_node    i_hash;          /* hash list */
    struct list_head     i_list;          /* list of inodes */
    struct list_head     i_sb_list;       /* list of superblocks */
    struct list_head     i_dentry;        /* list of dentries */
    unsigned long         i_ino;           /* inode number */
    atomic_t              i_count;         /* reference counter */
    unsigned int          i_nlink;         /* number of hard links */
    uid_t                 i_uid;           /* user id of owner */
    gid_t                 i_gid;           /* group id of owner */
    kdev_t                i_rdev;         /* real device node */
    u64                   i_version;       /* versioning number */
    loff_t                i_size;          /* file size in bytes */
    seqcount_t            i_size_seqcount; /* serializer for i_size */
    struct timespec       i_atime;         /* last access time */
    struct timespec       i_mtime;         /* last modify time */
    struct timespec       i_ctime;         /* last change time */
    unsigned int          i_blkbits;       /* block size in bits */
    blkcnt_t              i_blocks;        /* file size in blocks */
    unsigned short         i_bytes;         /* bytes consumed */
    umode_t               i_mode;          /* access permissions */
    spinlock_t            i_lock;          /* spinlock */
    struct rw_semaphore    i_alloc_sem;    /* nests inside of i_sem */
    struct semaphore       i_sem;          /* inode semaphore */
    struct inode_operations *i_op;         /* inode ops table */
    struct file_operations *i_fop;         /* default inode ops */
    struct super_block     *i_sb;          /* associated superblock */
    struct file_lock       *i_flock;       /* file lock list */
    struct address_space    *i_mapping;     /* associated mapping */
    struct address_space    i_data;         /* mapping for device */
    struct dqquot           *i_dquot[MAXQUOTAS]; /* disk quotas for inode */
    struct list_head       i_devices;      /* list of block devices */
    union {
        struct pipe_inode_info *i_pipe;    /* pipe information */
        struct block_device     *i_bdev;    /* block device driver */
        struct cdev              *i_cdev;    /* character device driver */
    };
    unsigned long          i_dnotify_mask; /* directory notify mask */
    struct dnotify_struct   *i_dnotify;     /* dnotify */
    struct list_head        inotify_watches; /* inotify watches */
    struct mutex            inotify_mutex;   /* protects inotify_watches */
    unsigned long           i_state;         /* state flags */
    unsigned long           i_dirtied_when; /* first dirtying time */
    unsigned int            i_flags;         /* filesystem flags */
    atomic_t                i_writecount;    /* count of writers */
    void                   *i_security;     /* security module */
    void                   *i_private;      /* fs private pointer */
};

```

4. Wirtualny system plików (VFS).

- » c) struktura VFS
- » Inode object
- » - operacje na **inode** zawarte w strukturze **inode_operations**
- »
- »

```
struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,int,dev_t);
    int (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *,int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    void (*truncate_range)(struct inode *, loff_t, loff_t);
    long (*fallocate)(struct inode *inode, int mode, loff_t offset,
                      loff_t len);
    int (*fiemap)(struct inode *, struct fiemap_extents_info *, u64 start,
                  u64 len);
};
```

4. Wirtualny system plików (VFS).

- » **c) struktura VFS**
- » **Dentry object**
- » - reprezentuje tzw. **Directory entry** czyli komponent ścieżki
- » - nie jest to do końca faktyczny obiekt katalogu
- » - innymi słowy, **dentry** nie jest tym samym co katalog, a katalog sam w sobie jest innym rodzajem pliku
- » - operacje na **dentry** zawarte w strukturze **dentry_operations**

»

4. Wirtualny system plików (VFS).

» c) struktura VFS

» Dentry object

»

```

struct dentry {
    atomic_t          d_count;      /* usage count */
    unsigned int      d_flags;      /* dentry flags */
    spinlock_t        d_lock;      /* per-dentry lock */
    int               d_mounted;    /* is this a mount point? */
    struct inode       *d_inode;    /* associated inode */
    struct hlist_node  d_hash;      /* list of hash table entries */
    struct dentry      *d_parent;   /* dentry object of parent */
    struct qstr        d_name;      /* dentry name */
    struct list_head   d_lru;       /* unused list */
    union {
        struct list_head d_child;   /* list of dentries within */
        struct rcu_head  d_rcu;     /* RCU locking */
    } d_u;
    struct list_head   d_subdirs;   /* subdirectories */
    struct list_head   d_alias;     /* list of alias inodes */
    unsigned long      d_time;      /* revalidate time */
    struct dentry_operations *d_op; /* dentry operations table */
    struct super_block  *d_sb;      /* superblock of file */
    void               *d_fsdata;   /* filesystem-specific data */
    unsigned char       d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
};

```

4. Wirtualny system plików (VFS).

» **c) struktura VFS**

» **Dentry object**

»

- operacje na **dentry**
zawarte w strukturze
dentry_operations

```
struct dentry_operations {  
    int (*d_revalidate) (struct dentry *, struct nameidata *);  
    int (*d_hash) (struct dentry *, struct qstr *);  
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);  
    int (*d_delete) (struct dentry *);  
    void (*d_release) (struct dentry *);  
    void (*d_iput) (struct dentry *, struct inode *);  
    char *(*d_dname) (struct dentry *, char *, int);  
};
```

»

4. Wirtualny system plików (VFS).

- » **c) struktura VFS**
- » **File object**
- » - reprezentuje otwarty plik, który jest powiązany z procesem
- » - operacje na **file** zawarte w strukturze **file_operations**

- »

4. Wirtualny system plików (VFS).

» c) struktura VFS

» File object

»

»

```
struct file {
    union {
        struct list_head fu_list;      /* list of file objects */
        struct rcu_head fu_rcuhead;    /* RCU list after freeing */
    } f_u;
    struct path f_path;                /* contains the dentry */
    struct file_operations *f_op;      /* file operations table */
    spinlock_t f_lock;                 /* per-file struct lock */
    atomic_t f_count;                  /* file object's usage count */
    unsigned int f_flags;              /* flags specified on open */
    mode_t f_mode;                     /* file access mode */
    loff_t f_pos;                      /* file offset (file pointer) */
    struct fown_struct f_owner;        /* owner data for signals */
    const struct cred *f_cred;         /* file credentials */
    struct file_ra_state f_ra;         /* read-ahead state */
    u64 f_version;                     /* version number */
    void *f_security;                  /* security module */
    void *private_data;                /* tty driver hook */
    struct list_head f_ep_links;       /* list of epoll links */
    spinlock_t f_ep_lock;              /* epoll lock */
    struct address_space *f_mapping;    /* page cache mapping */
    unsigned long f_mnt_write_state;   /* debugging state */
};
```

4. Wirtualny system plików (VFS).

- » c) struktura VFS
- » File object
- » - operacje na file zaw
w strukturze
file_operations

»

```

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                        unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                        unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
                unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *,
                        int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *,
                                        unsigned long,
                                        unsigned long,
                                        unsigned long,
                                        unsigned long);

    int (*check_flags) (int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write) (struct pipe_inode_info *,
                            struct file *,
                            loff_t *,
                            size_t,
                            unsigned int);
    ssize_t (*splice_read) (struct file *,
                            loff_t *,
                            struct pipe_inode_info *,
                            size_t,
                            unsigned int);
    int (*setlease) (struct file *, long, struct file_lock *);
};

```

4. Wirtualny system plików (VFS).

- » c) struktura VFS
- » **PODSUMOWANIE:**
- » **Wirtualny System Plików** to warstwa pośrednicząca między aplikacją a systemem plików. Jej celem jest umożliwienie aplikacjom dostępu do różnych rodzajów konkretnych systemów plików w jednolity sposób.
- » W powszechnym modelu plików każdy byt (katalog, urządzenie, plik wykonywalny) traktowany jest jako zwykły **plik**.
- » Cztery główne obiekty **VFS** to: **superblock**, **inode**, **dentry** oraz **file**.
- »

Kompilacja

Po co kompilować jądro?

- a) Kompilujemy jądro pod nasz procesor, w teorii powinno to zwiększyć wydajność systemu.
- b) Dostosowanie jądra do naszych potrzeb. Włączenie/wyłączenie modułów lub opcji.
- c) Poznanie możliwości jądra i przetestowanie nowych możliwości.
- d) Instalacja innej wersji jądra niż obecnie zainstalowana

Moduły

Moduły to dynamicznie podłączane elementy jądra(głównie sterowniki).

lsmod - listening modułów

modprobe - zarządzanie modułami(przyłączanie/odłączanie)

modinfo <nazwa> - informację na temat modułu

Moduły pozwalają na zaoszczędzeniu pamięci zajmowanej przez jądro, ładowanie jedynie potrzebnych elementów oraz ograniczanie wielkości jądra. Przy instalacji systemu, instalator wykrywa nasz sprzęt i instaluje automatycznie potrzebne moduły.

Kompilacja jądra

1. Pobranie kodu źródłowego jądra: <https://www.kernel.org/>
2. Wypakowanie archiwum
3. Konfiguracja pliku .config - *make menuconfig*
4. Instalacja potrzebnych programów -
apt-get install git fakeroot build-essential ncurses-dev xz-utils libssl-dev bc flex libelf-dev bison
5. Kompilacja jądra - *make* (może zająć parę godzin)
6. Instalacja modułów - *make modules_install*
7. Instalacja jądra - *make install*
8. *update-initramfs*

6. System calls.

- a) User mode VS Kernel mode
- b) czym jest System Call?
- c) komunikacja z jądrem systemu
- d) typy System Calli - dostępne wywołania systemowe

6. System calls.

a) User mode VS Kernel mode

Aby zrozumieć, czym dokładnie są system calls, dobrze jest najpierw zaznajomić się z dwoma trybami wykonywania operacji – **tryb użytkownika (user mode)** oraz **tryb jądra (kernel mode)**.

6. System calls.

a) User mode VS Kernel mode

- **user mode** oraz **kernel mode** są dwoma trybami, w których program może się wykonywać
- program wykonujący się w trybie użytkownika **nie ma** bezpośredniego dostępu do zasobów, pamięci czy sprzętu
- program wykonujący się w trybie jądra **ma** bezpośredni dostęp do zasobów, pamięci oraz sprzętu
- systemy dostarczają **interfejs** do usług oferowanych przez **system operacyjny**.

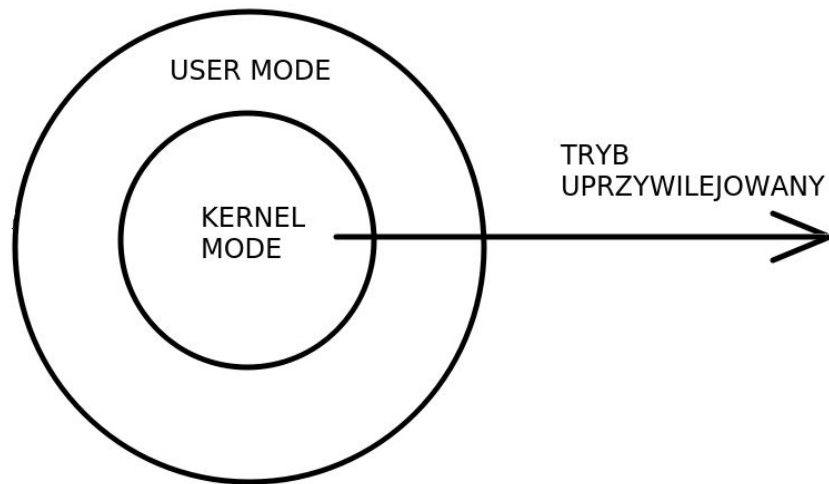
6. System calls.

a) User mode VS Kernel mode

Problem:

Praca programu w trybie uprzywilejowanym może mieć fatalne skutki.

Dlaczego?

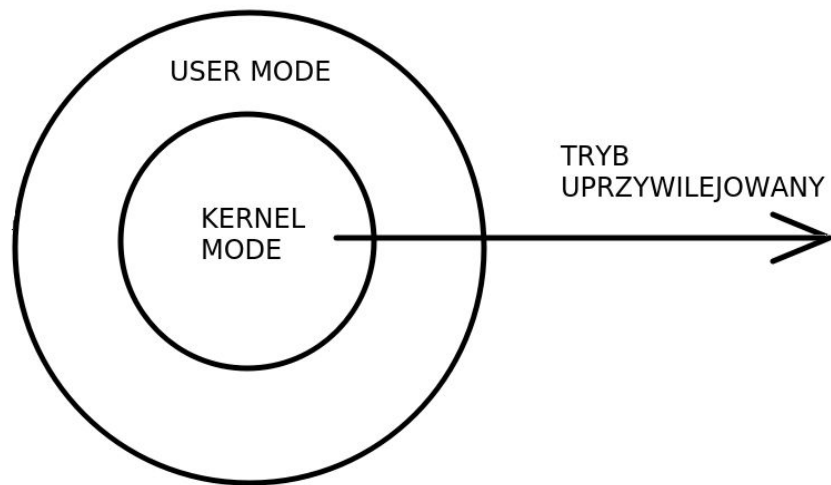


Programowanie
niskopoziomowe

6. System calls.

a) User mode VS Kernel mode

Jeżeli do programu, operującego w danym momencie w trybie uprzywilejowanym dostanie się awaria - cały system się zawiesi i wstrzyma swoje działanie !

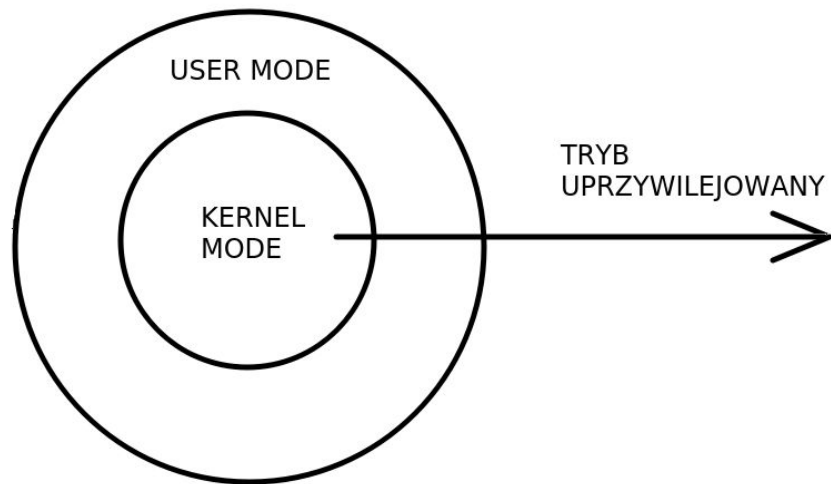


6. System calls.

a) User mode VS Kernel mode

Wniosek:

Praca w trybie użytkownika jest bezpieczniejsza niż praca w trybie jądra.



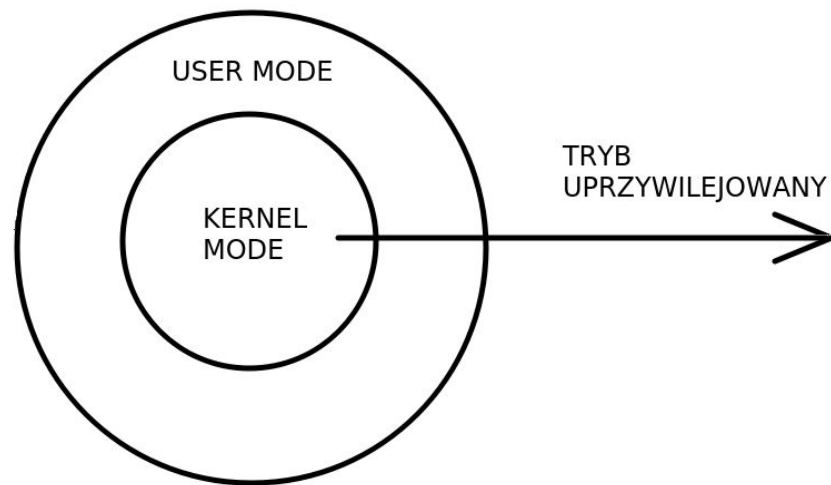
6. System calls.

a) User mode VS Kernel mode

Dlatego też większość programów wykonuje się w trybie użytkownika - ponieważ jest on bezpieczny.

Ale...

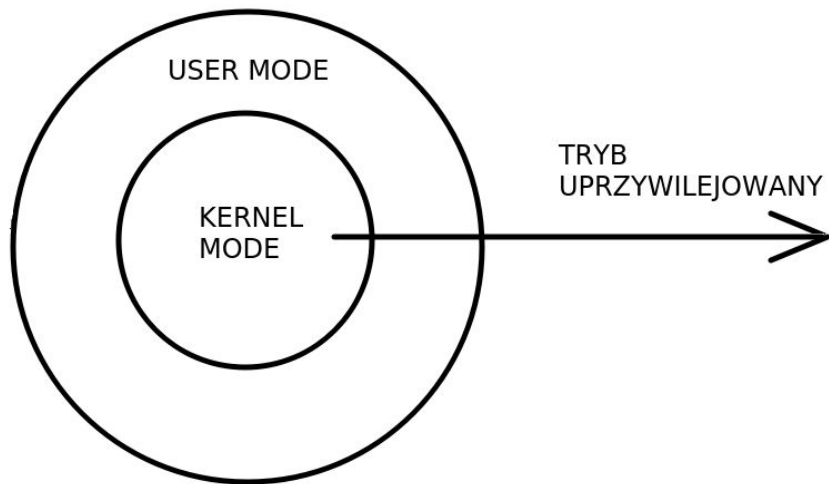
Program wykonujący się w trybie użytkownika czasami potrzebuje dostępu do niektórych zasobów lub pamięci.



6. System calls.

a) User mode VS Kernel mode

Gdy program potrzebuje do nich dostępu, a działa w trybie użytkownika – wykonuje wywołanie do systemu operacyjnego w celu prośby o udostępnienie prawa do zasobów. W tym momencie program zmienia tryb z **trybu użytkownika** w **tryb jądra**, dzięki czemu może używać tych zasobów. Nazywa się to **mode switching**.

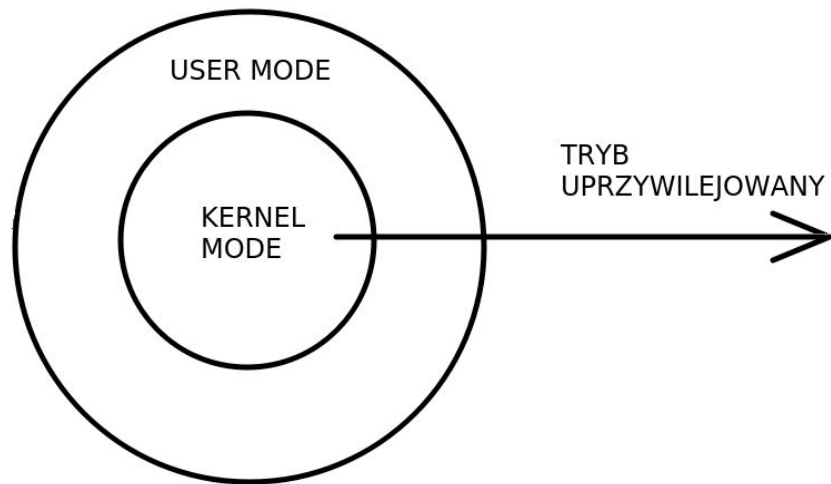


6. System calls.

b) czym jest System Call?

Takie właśnie wywołanie programu o dostęp do zasobów nazywa się **System call'em**.

System call jest wywoływany przez program, kiedy potrzebuje on dostępu do pamięci, hardware'u czy do innych pewnych zasobów.

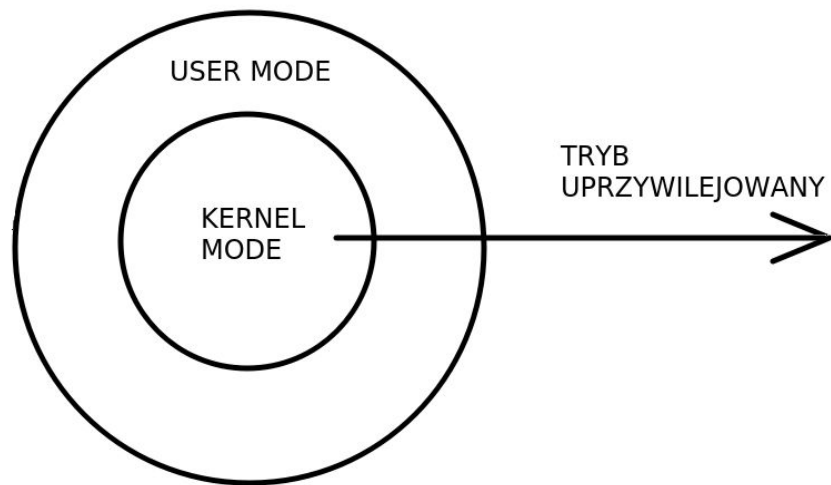


6. System calls.

b) czym jest System Call?

Innymi słowy można powiedzieć, że **System call** to programowalny sposób, w którym program komputerowy żąda usługi z jądra systemu operacyjnego.

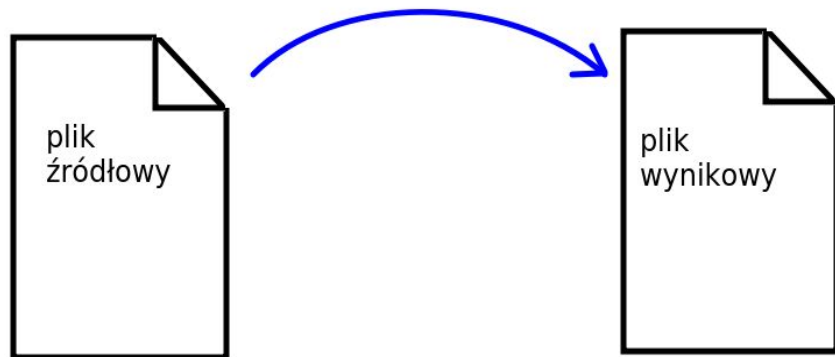
Takie wywołania generalnie są dostępne jako metody napisane w językach C oraz C++.



6. System calls.

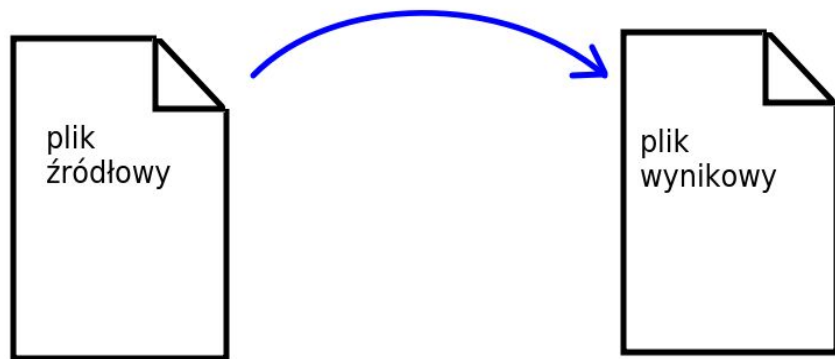
c) komunikacja z jądrem systemu

Przykład:



Sekwencja wywołań systemowych podczas działania prostego programu wypisującego zawartość pliku a następnie kopiującego jego zawartość do pliku wynikowego.

6. System calls.



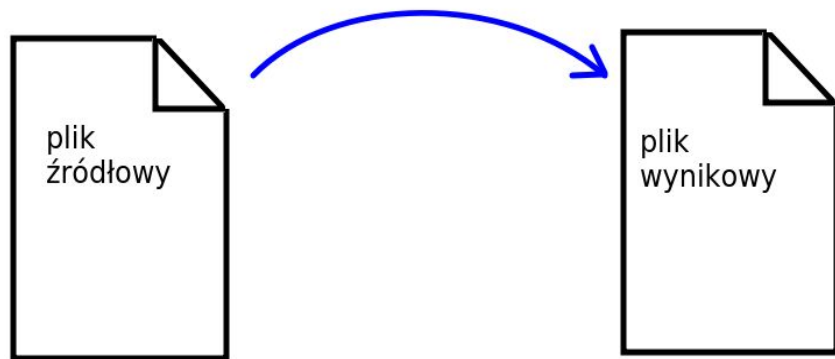
c) komunikacja z jądrem systemu

Nabycie nazwy
pliku źródłowego

Wypisanie komunikatu

Zaakceptowanie wejścia
- klawiatura, myszka itp.

6. System calls.



c) komunikacja z jądrem systemu

Nabycie nazwy
pliku źródłowego

Wypisanie komunikatu

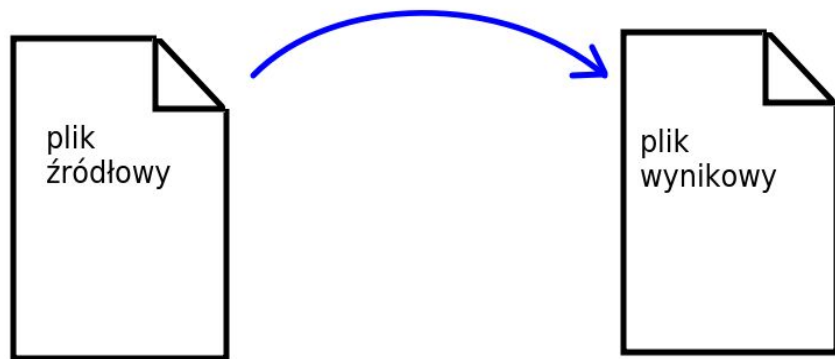
Zaakceptowanie wejścia
- klawiatura, myszka itp.

Nabycie nazwy
pliku wynikowego

Wypisanie komunikatu

Zaakceptowanie wejścia
- klawiatura, myszka itp.

6. System calls.



c) komunikacja z jądrem systemu

Nabycie nazwy
pliku źródłowego

Anulowanie, jeśli
plik nie istnieje

Wypisanie komunikatu

Zaakceptowanie wejścia
- klawiatura, myszka itp.

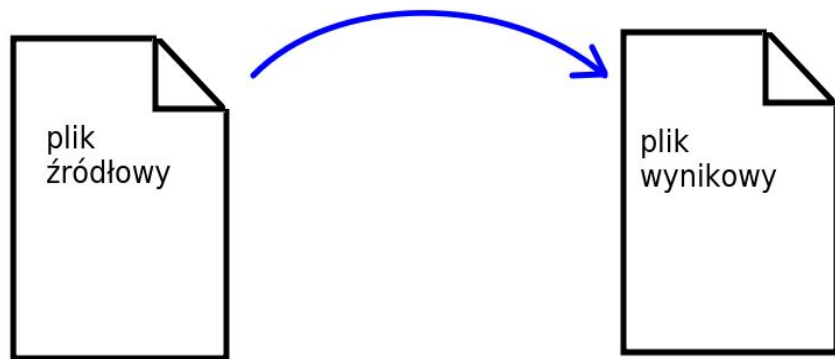
Nabycie nazwy
pliku wynikowego

Wypisanie komunikatu

Zaakceptowanie wejścia
- klawiatura, myszka itp.

Otworzenie pliku
wejściowego

6. System calls.



c) komunikacja z jądrem systemu

Nabycie nazwy pliku źródłowego

Wypisanie komunikatu

Zaakceptowanie wejścia
- klawiatura, myszka itp.

Nabycie nazwy pliku wynikowego

Wypisanie komunikatu

Zaakceptowanie wejścia
- klawiatura, myszka itp.

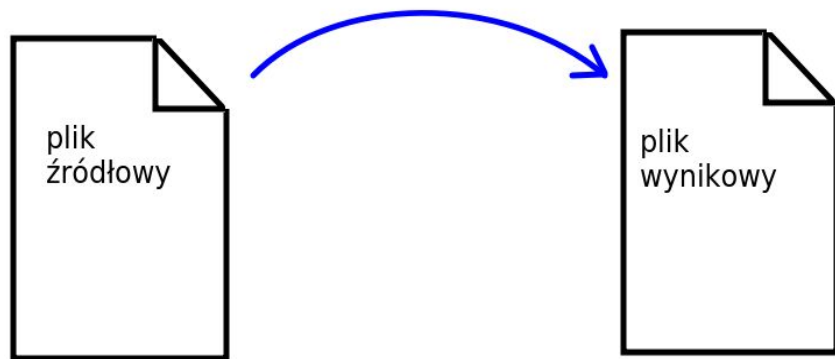
Otworzenie pliku wejściowego

Anulowanie, jeśli plik nie istnieje

Stworzenie pliku wynikowego

Jeżeli plik wynikowy istnieje - anulowanie

6. System calls.



c) komunikacja z jądrem systemu



Programowanie
niskopoziomowe

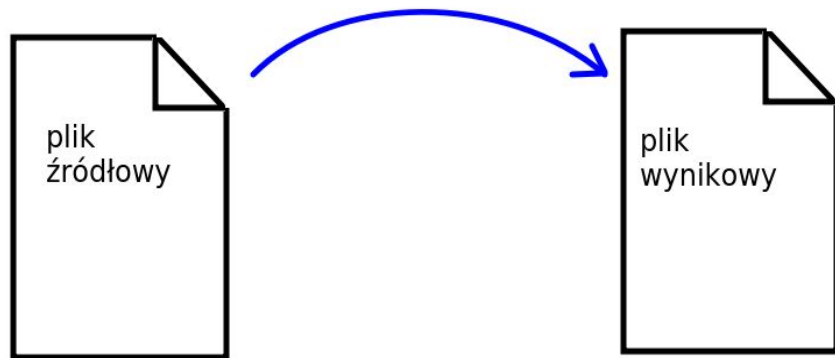
6. System calls.

c) komunikacja z jądrem systemu

Obserwacja:

Nawet dla tak prostego przykładu wykonywanych jest mnóstwo **System Calli**.

Z reguły w ciągu jednej sekundy wykonywanych jest tysiące wywołań systemowych.



6. System calls.

d) typy System Calli - dostępne wywołania systemowe

Typy system calli można podzielić na pięć głównych kategorii:

1. Kontrola procesów.
2. Manipulacja plikami.
3. Zarządzanie urządzeniami.
4. Utrzymywanie informacji systemowych.
5. Komunikacja.

6. System calls.

d) typy System Calli - dostępne wywołania systemowe

1. Kontrola procesów.

- kończenie, anulowanie;
- wczytanie, egzekucja;
- alokacja i zwalnianie pamięci;
- stworzenie i kończenie procesów,
- pobieranie atrybutów procesów;
- ustawianie atrybutów procesów;
- oczekiwanie na egzekucję procesu;
- oczekiwanie na zdarzenie;

6. System calls.

d) typy System Calli - dostępne wywołania systemowe

2. Manipulacja plikami.

- tworzenie i usuwanie;
- otwieranie i zamykanie;
- odczytywanie i zapisywanie;
- pobieranie atrybutów pliku;
- ustawianie atrybutów pliku;

6. System calls.

d) typy System Calli - dostępne wywołania systemowe

3. Zarządzanie urządzeniami.

- żądanie urządzenia, zwolnienie urządzenia;
- czytanie, zapisywanie;
- pobieranie atrybutów urządzenia;
- ustawianie atrybutów urządzenia;
- logiczne podłączanie i odłączanie urządzeń;

6. System calls.

d) typy System Calli - dostępne wywołania systemowe

4. Utrzymywanie informacji systemowych.

- pobranie i ustawienie czasu lub daty;
- pobranie i ustawienie danych systemowych;
- pobranie i ustawienie atrybutów procesów, plików lub urządzeń

6. System calls.

d) typy System Calli - dostępne wywołania systemowe

5. Komunikacja.

- tworzenie i usuwanie połączeń komunikacyjnych;
- przesyłanie i odbieranie wiadomości;
- transfer statusu informacji;
- dołączanie i odłączanie zdalnych urządzeń;

6. System calls.

Podsumowanie:

- programy mogą wykonywać się w dwóch trybach: **użytkownika** oraz **jądra**,
- **tryb jądra** jest trybem uprzywilejowanym – ma dostęp do pamięci, urządzeń oraz zasobów systemu operacyjnego,
- **wywołanie systemowe (system call)** stanowi interfejs między wykonywanym programem a jądrem systemu operacyjnego,
- w ciągu sekundy wykonywanych jest tysiące **wywołań systemowych**

Źródła:

- 1 - Polski i Angielski artykuł o jądrze systemu na wikipedii(głównie rysunki i schematy)
- 2 - "Professional Linux Kernel Architecture" - Wolfgang Maurer
- 3 - <https://www.tutorialspoint.com/ipc-using-message-queues>
- 4 - <https://kernel.org>

DZIĘKUJEMY ZA UWAGĘ