

2.1 Definicje

Program – kod realizujący wybrane przez użytkownika (lub system) algorytmy

Zadanie – instancja programu w systemie

Proces – pojedyncza (najczęściej jedyna) wykonywana część programu, posiadająca własny blok kontrolny (zestaw rejestrów, wskaźnik instrukcji, stos, uchwyt zasobów...), rezydująca we własnym obszarze pamięci

Wątek – elementarna jednostka wykonania, działająca tylko w obrębie posiadającego ją procesu. Proces może mieć kilka wątków

wskaźnik	stan	PID	IP	rejstry (AX,BX,CX,DX,...)	inf. o pamięci	uchwyty	
----------	------	-----	----	---------------------------	----------------	---------	-------

Zarządzanie procesami przez system operacyjny:

- utworzenie procesu,
- wstrzymywanie wykonywania,
- zakończenie procesu,
- przydział zasobów: procesora, pamięci, sprzętu, systemu plików,
- synchronizacja poprzez obiekty synchronizacyjne,
- komunikacja międzyprocesowa.

Wątki tworzone są w działającym procesie.

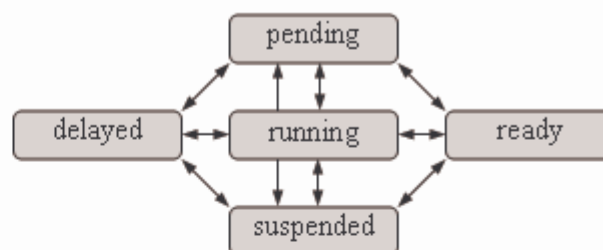
Zakończenie działania wątków może nastąpić w sposób naturalny, poprzez akcję procesu lub wraz z zamykaniem procesem.

Proces powołujący do działania inny proces nazywamy procesem macierzystym, zaś powoływany – potomnym.

Proces potomny, który utracił proces macierzysty, nazywamy procesem sierocym.

Stany procesu/wątku:

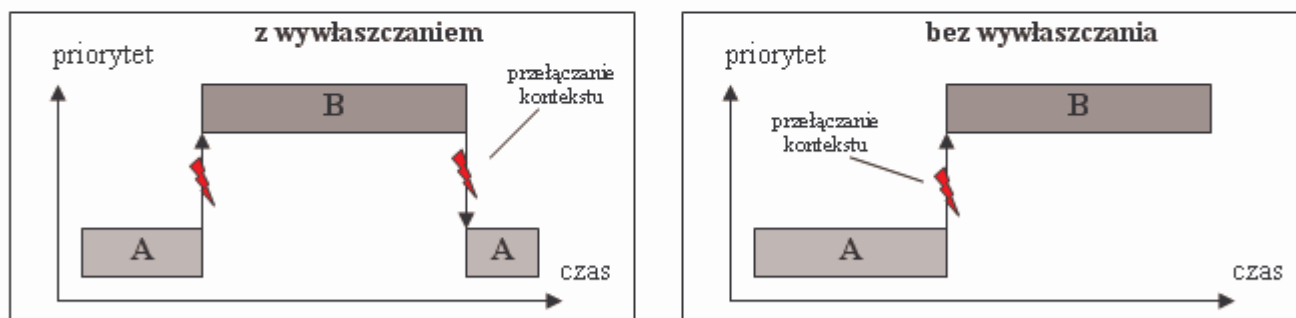
- wykonywany (running)
- gotowy do wykonania (ready)
- oczekujący na zasób (pending)
- zawieszony (suspended)
- oczekujące przez pewien czas (delayed)



Równoległe wykonanie procesów

Współbieżne wykonanie procesów

Wywłaszczanie procesu (preemption) i brak wywłaszczania

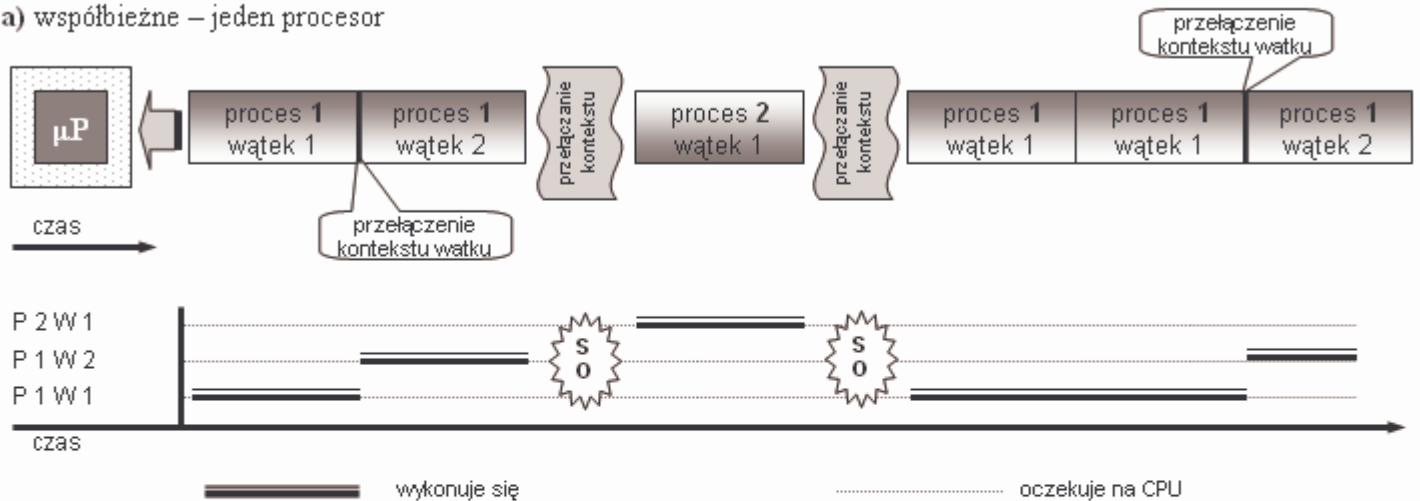


Sytuacje problematyczne:

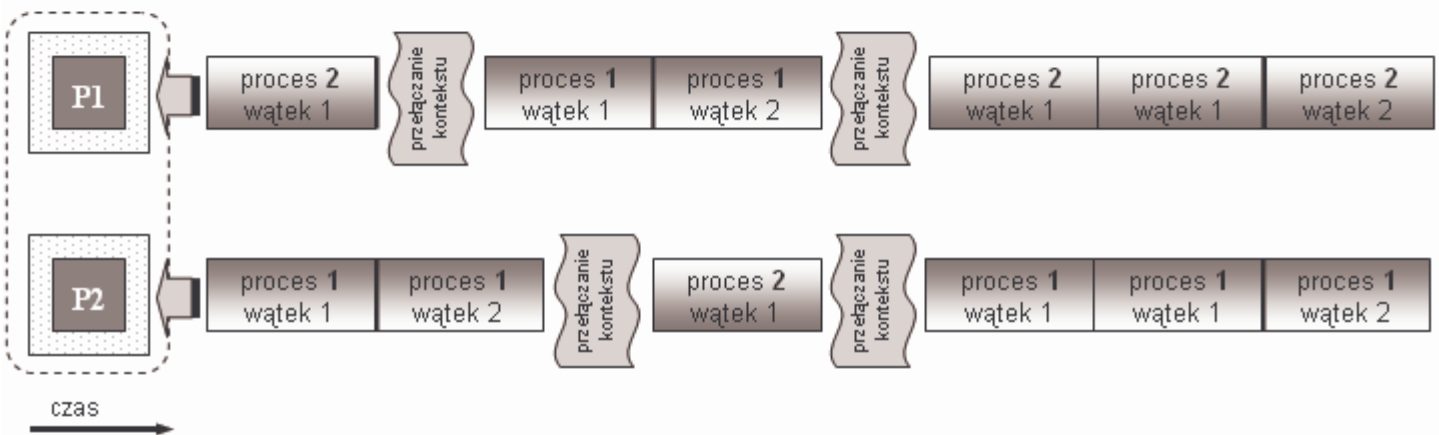
- zakleszczenie - problem zależny od oprogramowania, omawiany przy okazji synchronizacji procesów
- przeterminowanie/ zagłodzenie (starvation) – różne rozwiązania (lub brak rozwiązań)
- inwersja priorytetów

Wątki i procesy w systemie - wykonanie

a) współbieżne – jeden procesor



b) współbieżne, równoległe – dwa procesory (lub więcej)



Wątki w systemie – programowanie

```
// WĄTKI - LINUX

#include <pthread.h>

// funkcja (ciało) wątku

void* watek(void* v)
{
    // ... KOD WĄTKU
    // jednorazowy lub zapętłony

    return NULL;
}

// funkcja główna main

int main()
{
    pthread_t ptid;
    pthread_create(&ptid, NULL, &watek, NULL);

    // ... kod programu
    return 0;
}
```

```
// WĄTKI - WINDOWS

#include <windows.h>

// funkcja (ciało) wątku

unsigned long __stdcall watek(void *v)
{
    // ... KOD WĄTKU
    // jednorazowy lub zapętłony

    return 0;
}

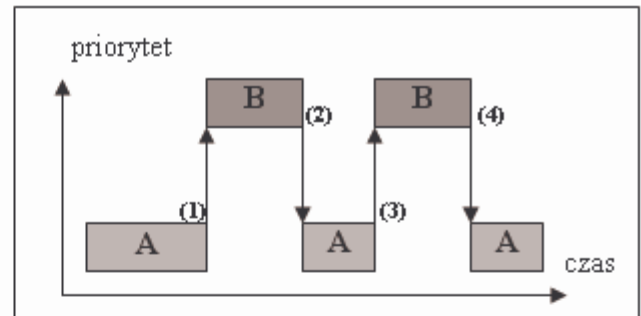
// funkcja główna main

int main()
{
    HANDLE hw;
    hw = CreateThread(0,0,watek,0,0,0);

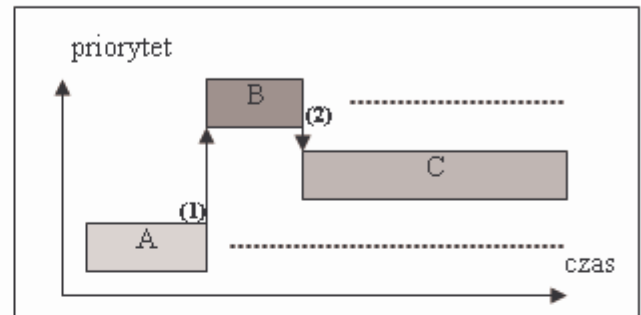
    // ... kod programu
    CloseHandle(hw);
    return 0;
}
```

Problem inwersji priorytetów

- (1) - pojawia się proces B gotowy do wykonania, proces A zostaje wywłaszczony
- (2) - proces B żąda dostępu do zasobu zajętego przez A
- (3) - A oddaje zasób, system automatycznie wywłaszcza A
- (4) - B kończy działanie, A przejmuję zasób procesora



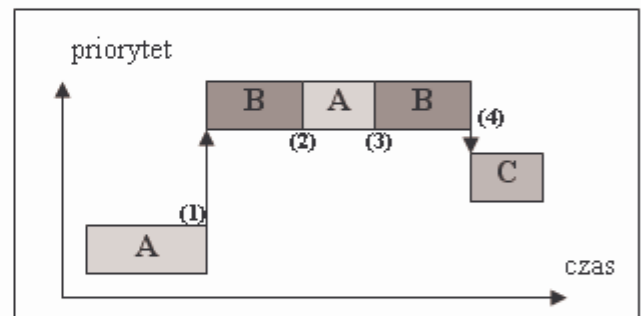
- (1) - pojawia się proces B gotowy do wykonania, proces A zostaje wywłaszczony
- (2) - proces B żąda dostępu do zasobu zajętego przez A, zostaje zatrzymany, wykonywany zostaje proces C, który blokując proces A uniemożliwia oddanie zasobu



Rozwiązania problemu inwersji priorytetów:

- inne rozwiązania (pośrednie, np.: obrona przed zagłodzeniem)
- dziedziczenie priorytetu

- (1) - pojawia się proces B gotowy do wykonania, proces A zostaje wywłaszczony
- (2) - proces B żąda dostępu do zasobu zajętego przez A, system znajduje proces posiadający zasób i nadaje mu priorytet procesu B
- (3) - A oddaje zasób, system automatycznie przywraca A poprzedni priorytet i wywłaszcza
- (4) - B kończy działanie, C przejmuję zasób procesora

**2.2 Wyznaczanie kolejności wykonywania procesów (szeregowanie procesów)****Algorytm FIFO (First In, First Out)**

W systemie aktywują się wątki P_1, P_2, P_3, P_4



Kolejność wykonania = kolejność nadejścia.

Algorytm SJF (Shortest Job First) także jako SPT (Shortest Processing Time)

W systemie aktywują się wątki P_1, P_2, P_3, P_4

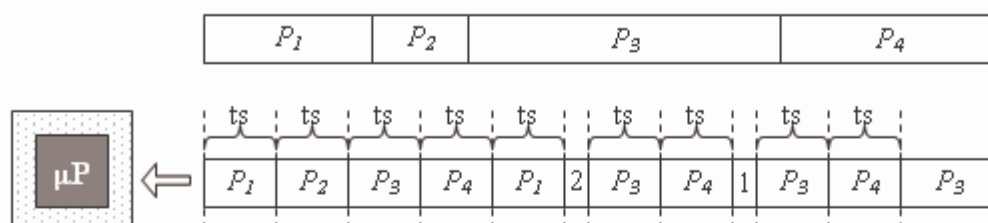


Kolejność wykonania: od najkrótszego do najdłuższego.

Algorytm karuzelowy (Round Robin)

W systemie przydziela się każdemu zadaniu zasób procesora na czas nie większy niż z góry określony przedział (nazywany też *quantum* albo *time slice*). Po jego przeminięciu zadanie jest wywłaszczane i przesuwane na koniec kolejki, jeśli w niej są inne zadania.

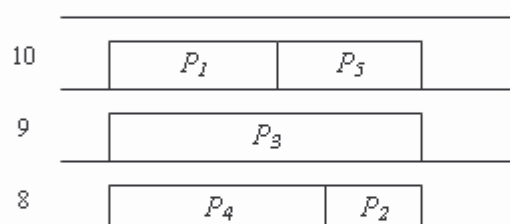
W systemie aktywują się wątki P_1, P_2, P_3, P_4



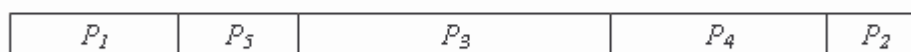
Algorytmy wykorzystujące priorytety

Kolejka priorytetowa

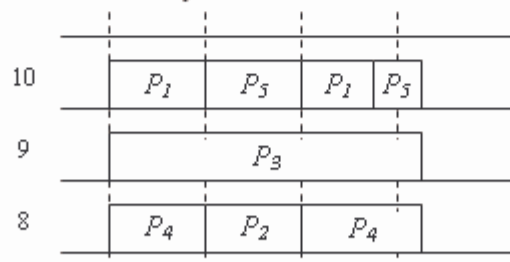
Wątki są wstawiane do odpowiednich kolejek na podstawie posiadanego priorytetu. W obrębie kolejki obowiązuje kolejność FIFO, zaś system zaczyna wykonywanie od kolejki o najwyższym priorytecie.



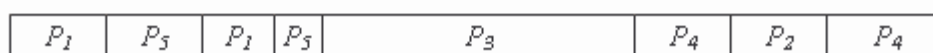
Kolejność wykonania:



Kolejka priorytetowa z algorytmem karuzelowym



Kolejność wykonania:



Algorytm EDF (Earliest Deadline First)

EDF polega na nadaniu najwyższego priorytetu zadaniu, które ma najwcześniejszy dopuszczalny termin wykonania, przy czym nie jest uwzględniany czas wykonania tego zadania.

Algorytm MLF (Minimum Laxity First)

MLF polega na nadaniu najwyższego priorytetu zadaniu, które ma najwcześniejszy dopuszczalny czas rozpoczęcia. Dopuszczalny czas rozpoczęcia to różnica terminu wykonania i czasu wykonania zadania.

Algorytm dynamicznego przydziału priorytetu EDF i MLF wymagają znajomości (możliwości określenia) dopuszczalnego terminu wykonania zadania, dodatkowo MLF wymaga możliwości precyzyjnego wyznaczania czasu wykonania zadania.

Algorytm RMS (Rate Monotonic Scheduling)

RMS polega na przypisaniu zadaniom stałego i unikalnego priorytetu. Jego wartość zależy od okresu pojawiania się tych zadań w systemie – im okres jest krótszy, tym priorytet jest wyższy. Zakłada się, że zadanie o wyższym priorytecie wywłaszcza zadanie o niższym priorytecie. Jako termin wykonania zadania przyjmuje się termin ponownego wznowienia tego zadania (czyli długość okresu jego aktywacji).

Warunek konieczny RMS [Lin73]

Niech $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ będzie zbiorem niezależnych i okresowych zadań o czasach wykonania c_i i okresach t_i , uporządkowanych według nierosnących priorytetów. Dla zadania τ_i wszystkie zadania muszą zostać zakończone w przedziale $t_i - c_i$. W trakcie tego przedziału t_i każde zadanie o wyższym priorytecie τ_j może mieć liczbę x aktywacji daną wzorem (4.1):

$$x = \left\lceil \frac{t_i}{t_j} \right\rceil \quad (4.1)$$

i zajmuje zasób przez czas dany wzorem (4.2):

$$T_j = c_j \left\lceil \frac{t_i}{t_j} \right\rceil \quad (4.2)$$

Czas Y_i , po jakim się wykonane zostanie zadanie τ_i , dla najgorszego przypadku (wszystkie zadania aktywują się naraz) jest sumą czasu jego wykonania oraz czasów wykonania wszystkich zadań o wyższym priorytecie (4.3):

$$Y_i = \sum_{j=1}^i c_j \left\lceil \frac{t_i}{t_j} \right\rceil \quad (4.3)$$

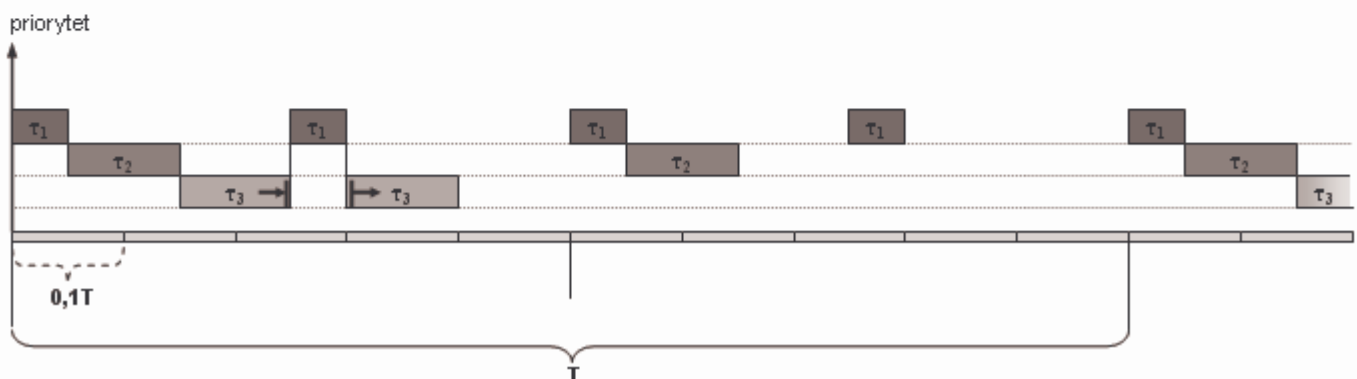
Dane zadanie τ_i spełnia ograniczenia, jeśli czas zakończenia jego wykonywania jest nie większy niż okres aktywacji. Wystarczającym wymogiem na dotrzymanie przez zadanie τ_i terminu jest spełnienie nierówności (4.4):

$$\sum_{j=1}^{i-1} \left(\left\lceil \frac{t_i}{t_j} \right\rceil c_j \right) \leq t_i - c_i \quad (4.4)$$

Powyższy warunek (4.4) służy do sprawdzenia dotrzymywania ograniczeń czasowych pojedynczego zadania τ_i . Jeśli wszystkie zadania dotrzymują terminów, mówi się, że zbiór zadań $\{\tau_1, \tau_2, \dots, \tau_n\}$ jest szeregowalny.

$\tau = \{\tau_1, \tau_2, \tau_3\}$:

$\tau_1 = (c_1, t_1) = (0,05T, 0,25T)$, $\tau_2 = (c_2, t_2) = (0,1T, 0,5T)$, $\tau_3 = (c_3, t_3) = (0,2T, T)$, $t_1 < t_2 < t_3 \Rightarrow p_1 > p_2 > p_3$



2.3 Szeregowanie procesów w systemie VxWorks

VxWorks (Wind River Systems) to system czasu rzeczywistego, oparty na architekturze mikrokernels.

- zakres priorytetów 0-255
- dziedziczenie priorytetów jako zabezpieczenie przed inwersją priorytetów
- dostępne algorytmy szeregowania:
 - kolejka priorytetowa
 - kolejka priorytetowa z alg. karuzelowym
- możliwość wyłączenia wywłaszczania zadań.
- zgodność zagadnień szeregowania zadań ze standardem POSIX (1003.1b)

2.4 Szeregowanie procesów w systemie QNX

System QNX (Quantum) jest systemem czasu rzeczywistego, dedykowanym głównie na platformę sprzętową PC. Jądro oparte jest na architekturze mikrokernels.

- 32/64 poziomy priorytetów (0-najniższy, 63 najwyższy)
- wywłaszczanie zadań

Procesy uruchomione z powłoki mają priorytet 10.

Dostępne są następujące algorytmy szeregowania zadań:

- FIFO,
- kolejka priorytetowa z alg. karuzelowym,
- algorytm sporadyczny,
- algorytm adaptacyjny.

Algorytm sporadyczny związany jest z dynamiczną zmianą wartości priorytetu, oscylującego dla procesów między wartościami aplikacji pierwszoplanowych i aplikacjami tła.

Algorytm adaptacyjny jest oparty na kolejce priorytetowej z alg. karuzelowym, przy czym została wprowadzona następująca modyfikacja:

- po zakończeniu wykonywania przez pełny przedział (time slice) priorytet procesu zmniejszany jest o 1. Jeśli po upływie określonego przedziału czasu proces nie dostanie zasobu procesora – priorytet jest zwiększany o 1, przy czym nie może być przekroczony poziom pierwotny.

2.5 Szeregowanie procesów w systemie Linux

System Linux jest systemem wywodzącym się od Unixa. Jest wielozadaniowy, posiada ochronę pamięci. Wywłaszczanie zadań jest realizowane tylko dla procesów działających w trybie użytkownika – tryb jądra celem uproszczenia konstrukcji oraz minimalizacji narzutów nie posiada wywłaszczania.

W Linuxie procesy posiadają następujące atrybuty, związane z szeregowaniem:

- **rt_priority** – zakres 0-99 – wyznacza, w której kolejce koncepcyjnej zostanie umieszczony proces gotowy do wykonania
- **priority** – jest to atrybut powiązany z wartością nice procesu w systemie – standardowo zdefiniowaną na 200ms,
- **counter** – jest to liczba impulsów zegara (1imp.=10ms), jakie pozostały procesowi do zakończenia wykonywania w przydzielonym przedziale czasowym.

Proces potomny dziedziczy wymienione parametry priority i rt_priority po procesie macierzystym.

Dostępne są następujące algorytmy szeregowania:

- SCHED_FIFO,
- SCHED_RR,
- SCHED_OTHER.

Dwa pierwsze algorytmy zostały przedstawione wcześniej. Stosuje się je w przypadkach wymagających właściwości systemów czasu rzeczywistego (**rt_priority** > 0).

Algorytm **SCHED_OTHER** jest stosowany do procesów posiadających atrybut **rt_priority** równy 0. Szeregowanie odbywa się w oparciu o dynamicznie wyznaczany priorytet – na podstawie wartości **priority** przydzielany jest czas wykonania, odmierzany atrybutem **counter**. Zegar systemowy odmierza takty co 10 ms.

Algorytm szeregowania procesów w Linuxie

Szeregowanie zadań w Linuxie odbywa się na podstawie statycznego i dynamicznego priorytetu zadania.

Priorytet statyczny jest wartością stałą dla zadania, przechowywaną w polu **rt_priority**. System nie zmienia jej w trakcie działania, zmienić ją natomiast można programowo.

Priorytet dynamiczny wyznaczany jest przez system w oparciu o pozostały czas, odmierzany w polu **priority**. Polityka systemu przydziela większy priorytet (pierwszeństwo wykonania) zadaniom, które mają wyznaczony dłuższy czas wykonania w epoce.

Wykonywanie procesów w systemie odbywa się w pewnym cyklu, zwanym epoką. Na początku epoki szeregowane są wszystkie zadania o priorytecie dynamicznym i zostaje im wyznaczony czas wykonania w obrębie epoki na podstawie wartości **nice** procesu.

Ochrona przed zagłodzeniem

Wyliczanie nowego przydziału czasu dla procesu wzbogacone jest o mechanizm ‘postarzania’. Do nowej wartości dodawana jest połowa niewykorzystanego przez proces czasu w minionej epoce. Pozwala to na ‘wybicie się’ procesu, który był zagłodzony.

Współpraca z użytkownikiem (aplikacje interaktywne)

Linux w starszych wersjach jądra nie miał bezpośredniego powiązania interaktywnego charakteru procesu z szeregowaniem zadań. Dzięki mechanizmowi obrony przed zagłodzeniem, aplikacje zawieszające się oczekiwaniem na zasób (klawiatura, mysz) nie tracą niedokończonego impulsu zegara, zatem przy wyznaczaniu nowych przydziałów są postarzane, a co za tym idzie – ich priorytet wzrasta.

Od jądra 2.6 wprowadzono tzw. *interactive credit* – parametr zwiększany, jeśli aplikacja długo nie uzyskuje procesora i zmniejszany w przypadku częstego korzystania z CPU.

Szeregowanie wieloprocessorowe

Każdy procesor ma własną kolejkę zadań do wykonania. Co pewien okres (zależny od całkowitego obciążenia) jest wywoływany moduł równoważący kolejki, jeśli różnice planowanych obciążeń CPU są większe od 25%, to następuje przenoszenie zadań na procesory mniej obciążone. Istnieje możliwość przypisania preferencji wykonania procesu do konkretnego procesora.

2.6 Szeregowanie zadań w systemie RTLinux

W systemie RTLinux dostępny jest tryb wykonywania procesów: *realtime*. Pozwala on na wykonywanie wątków w trybie jądra i w jego przestrzeni adresowej. Programowe przełączanie kontekstu przyspiesza wykonywanie, ułatwiona jest też wymiana danych. Niestety, brak ochrony pamięci stwarza potencjalne niebezpieczeństwa. Kolejnym ograniczeniem jest stały rozmiar stosu.

Zaimplementowane algorytmy szeregowania zadań to RMS i EDF.

2.7 Szeregowanie procesów w systemie WinNT

WinNT – system wielozadaniowy z wywłaszczaniem, podstawową jednostką szeregowania zadań jest wątek; algorytm szeregowania oparty o algorytm karuzelowy. Procesy posiadają priorytet bazowy i priorytet dynamiczny.

Poziomy priorytetów

W WinNT procesy posiadają priorytet bazowy i priorytet wykonywania. System rozróżnia 32 poziomy priorytetów:

- wartość 0 - bezczynności systemu
- zakres 1-15 - priorytety dynamiczne (system może zmieniać procesom priorytety wykonywania w trakcie upływu czasu)
- zakres 16-31 – priorytety czasu rzeczywistego – system nie może zmieniać priorytetów wykonywania

Poziom priorytetu funkcji systemowych – 11 – uwaga! cache systemowe pracują na poziomie priorytetu 11

W systemie istnieją następujące klasy priorytetów procesów:

- `REALTIME_PRIORITY_CLASS` (wartość bazowa: 24)
- `HIGH_PRIORITY_CLASS` (wartość bazowa: 13)
- `ABOVE_NORMAL_PRIORITY_CLASS` (wartość bazowa: 10)
- `NORMAL_PRIORITY_CLASS` (wartość bazowa: 8)
- `BELOW_NORMAL_PRIORITY_CLASS` (wartość bazowa: 6)
- `IDLE_PRIORITY_CLASS` (wartość bazowa: 4)

Wątkom w procesie można dodatkowo modyfikować priorytety w odniesieniu do wartości bazowej procesu poprzez modyfikatory:

- `THREAD_PRIORITY_LOWEST` (-2)
- `THREAD_PRIORITY_BELOW_NORMAL` (-1)
- `THREAD_PRIORITY_NORMAL` (0)
- `THREAD_PRIORITY_ABOVE_NORMAL` (+1)
- `THREAD_PRIORITY_HIGHEST` (+2)

lub pozwalając na osiągnięcie skrajnych wartości danego zakresu:

- `THREAD_PRIORITY_IDLE` (1 albo 16)
- `THREAD_PRIORITY_TIME_CRITICAL` (15 albo 31)

Wątki nie są gotowe do wykonania, gdy:

- są stworzone jako zawieszone,
- zostały zawieszone,
- czekają na zasób (element synchronizacji)

Najczęstsze powody wywłaszczenia:

- upłynął przyzany okres procesora
- pojawił się wątek o wyższym priorytecie, gotowy do wykonania,
- bieżący wątek przechodzi w uśpienie na pewien czas (delay).

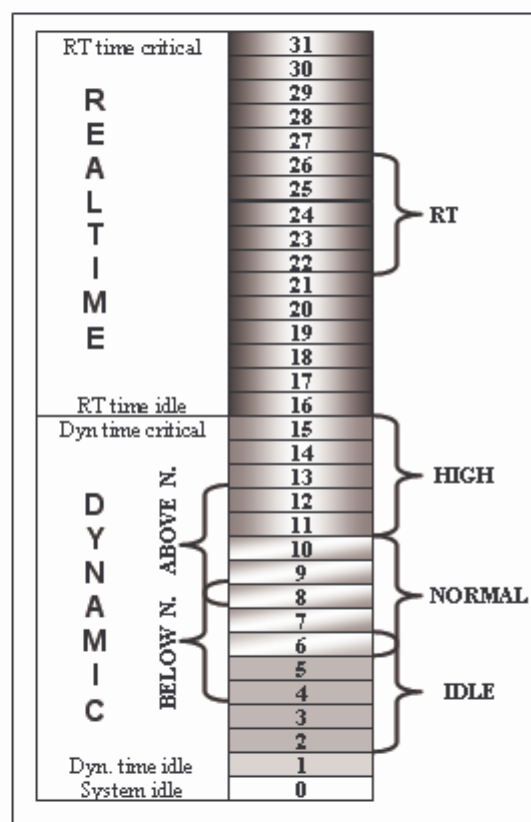
Zmiana kontekstu przebiega następująco:

- zapisywany jest kontekst wątku, którego wykonywanie zostało zakończone,
- odłożenie owego wątku na koniec jego kolejki priorytetowej,
- odszukanie gotowego do wykonania wątku, który ma najwyższy priorytet,
- usunięcie nowego wątku z kolejki priorytetowej.

Wartości przyznawanych przedziałów czasowych (time slice) procesora dla wątków:

- usług serwerowych – 120 ms,
- pierwszoplanowych – {60,40,20} ms, dodatkowo zwiększany priorytet o 1 pkt.
- pracujących w tle – 20 ms.

Długość przedziałów czasowych może być regulowana, podobnie wsparcie dla procesów pierwszoplanowych (odpowiedni wpis do rejestru).



Algorytm szeregowania procesów w systemie WinNT

W trakcie działania systemu scheduler działa w oparciu o następujące zasady:

- nadchodzące zadanie o wyższym priorytecie wywłaszcza (natychmiast) zadanie o niższym priorytecie,
- zadanie, które staje się gotowe do wykonania wywołuje funkcję *ReadyThread*,
- po zakończeniu się przedziału czasowego wykonywania bieżącego wątku lub po wstrzymaniu się wątku na obiekcie synchronizacji system wywołuje funkcję *FindReadyThread* aby wyznaczyć kolejny wątek do wykonywania,
- procesy, które otrzymały zasób mają podnoszony na pewien czas priorytet (boost+decay),
- co pewien dłuższy okres system sprawdza, czy nie ma procesów zagłodzonych – funkcja *ScanReadyQueues*

Funkcja *FindReadyThread*

Wyszukuje wątek gotowy do wykonania o najwyższym priorytecie.

Funkcja *ReadyThread*

Obsługuje wątki, które przeszły do stanu gotowości do wykonania. Jeżeli taki wątek ma priorytet wyższy, niż wątek aktualnie wykonywany – następuje wywłaszczenie, w przeciwnym przypadku umieszcza go w kolejce gotowych do wykonania. Przy okazji funkcja na początek owej kolejki wysuwa takie wątki, które zostały wydziedziczone przed ukończeniem przynajmniej jednego przyznanego przedziału czasu procesora (time slice).

Doładowanie procesu (*boosting + decay*)

Wątki o dynamicznym priorytecie mogą zostać "doładowane". Ma to miejsce w przypadku oczekiwania na pewien zasób. Wówczas priorytet wykonywania zostaje podniesiony (*boost*) o pewien poziom (np. zdarzenie pochodzące od klawiatury lub myszki - +5 (6?), zdarzenie związane z kartą sieciową, portem szeregowym, mailslotem, potokiem - 2, zdarzenie związane z wejściem/wyjściem HDD, LPT - 1, zdarzenie, uzyskanie obiektu semafora - 1). Nowe wartości nie mogą przekroczyć zakresu priorytetów dynamicznych. Po otrzymaniu zasobu wartość priorytetu wykonywania jest zmniejszana o 1 z każdym zakończeniem pełnego przedziału przydzielonego czasu procesora aż do osiągnięcia priorytetu bazowego (*decay*). W przypadku istnienia innego wątku o większym lub tym samym priorytecie, gotowego do wykonania, bieżący wątek jest wstawiany na koniec swojej kolejki priorytetowej (wg priorytetu wykonania). Kolejne oczekiwanie na zasób ponownie zwiększa priorytet wykonywania (aż do limitu -15).

Współpraca z użytkownikiem (aplikacje interaktywne)

Interaktywne wątki oczekujące na interakcje użytkownika (sygnał z klawiatury, myszki) są wspierane poprzez mechanizm doładowania procesu z najwyższymi możliwymi wartościami doładowania (p. poprzedni punkt). Ponadto procesy pierwszoplanowe mają podwyższony o 1 pkt priorytet.

Ochrona przed zagłodzeniem

Co 1s wywoływany jest Balance Set Manager (BSM), min. do sterowania pamięcią. Dodatkowo sprawdza on, czy nie ma wątków, które nie wykonywały się od 3s funkcją *ScanReadyQueues*. Jeśli takowy znajdzie – nadaje mu maksymalny priorytet zakresu (15) i wydłuża 2x przydzielony przedział czasu procesora (time slice). Po wykonaniu wątku przez zadany czas wartość priorytetu i długość przedziału wracają do stanu poprzedniego.

Inwersja priorytetów w systemie WinNT

W przypadku wątków o priorytecie bazowym REALTIME – system powierza programiście rozwiązanie problemu (uniknięcie blokady). W przypadku wątków o priorytecie z zakresu dynamicznego – problem blokady jest rozwiązywany przez algorytm ochrony przed zagłodzeniem.

Szeregowanie wieloprocessorowe

W WinNT procesy o najwyższym priorytecie są przydzielane do dostępnych procesorów. Jeden procesor jednak zostaje wydzielony i przy pełnym obciążeniu systemu ów procesor obsługuje pozostałe procesy o priorytetach niższych.

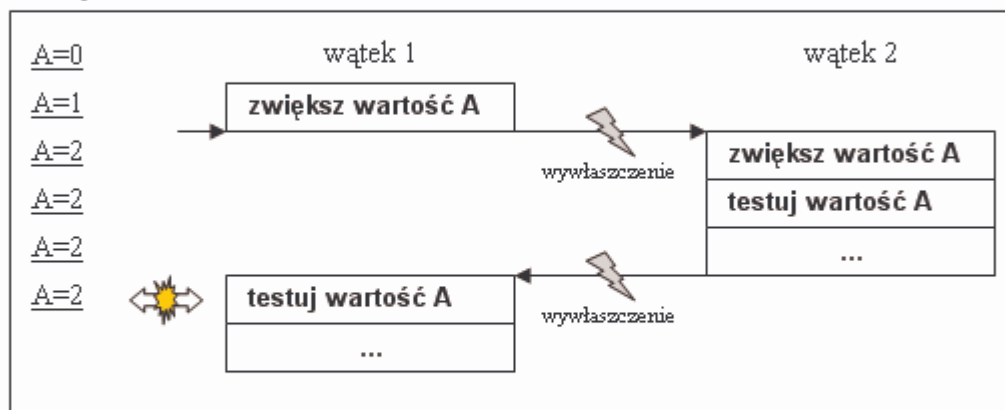
W systemie jest możliwość przypisywania procesów do konkretnego procesora. Jeżeli przypisany zbiór procesorów jest zajęty, to proces ustawia się w kolejce oczekujących nawet, jeśli inne procesory będą wolne.

3.1 Instrukcje atomowe

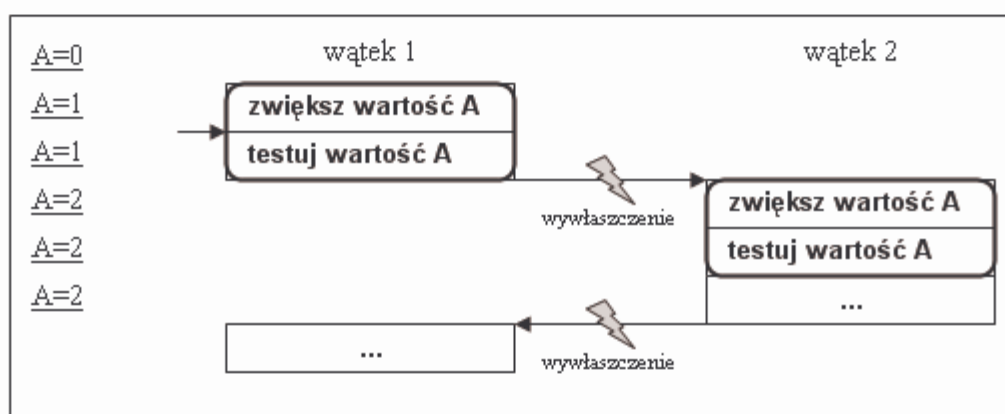
atrybut *volatile* – wymusza na kompilatorze korzystanie z aktualnej wartości zmiennej (omija cache (optymalizacja))

Przykład 1- błąd testowania

Bez funkcji atomowych

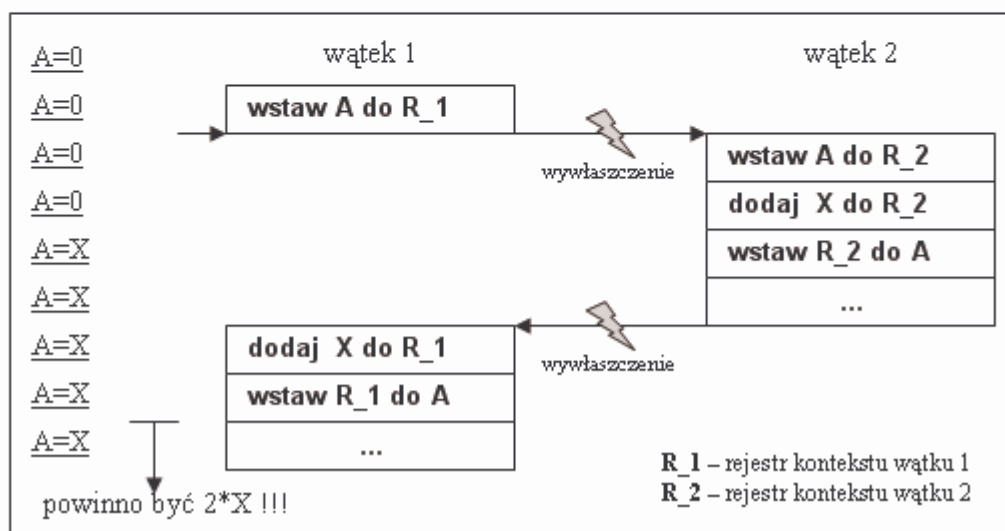


Z funkcjami atomowymi

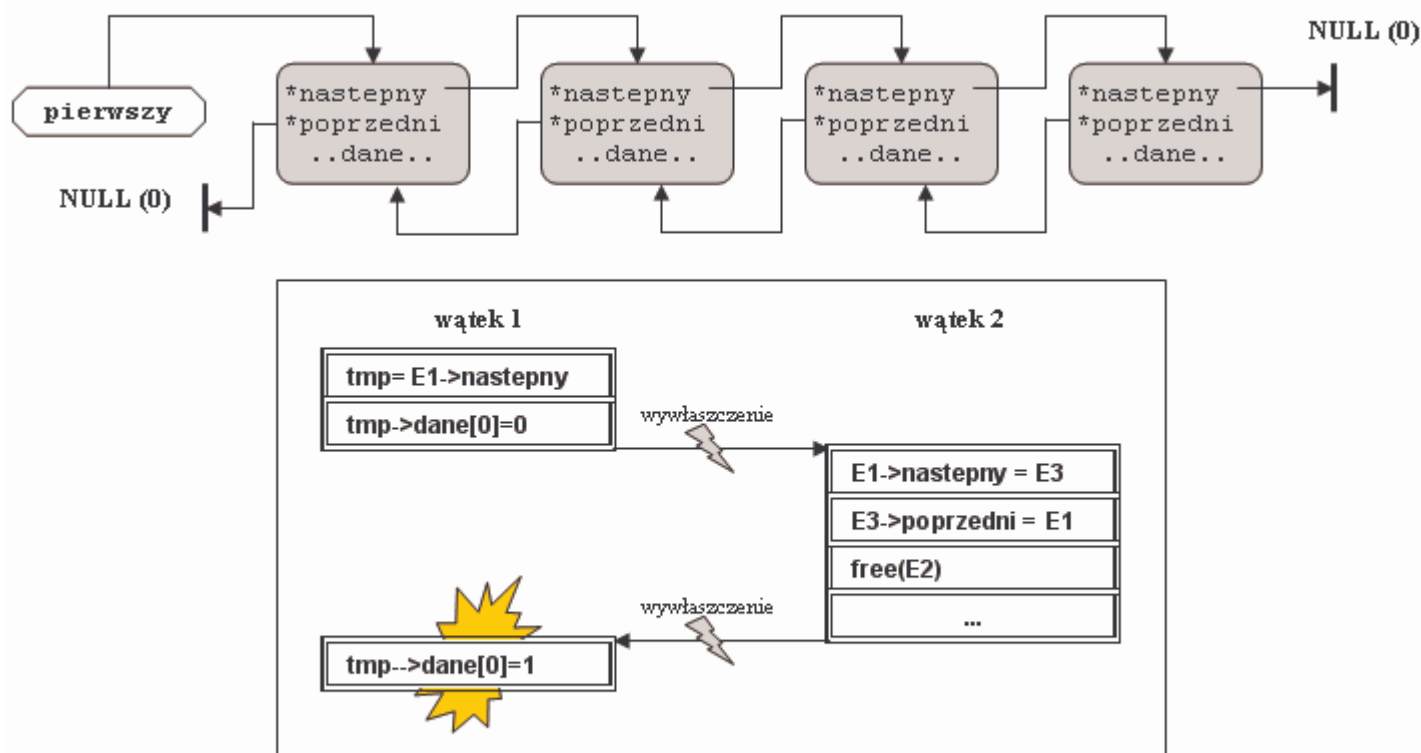


Przykład 2 – błąd aktualizacji

dodawanie liczb z wykorzystaniem rejestru



3.2 Sekcje krytyczne



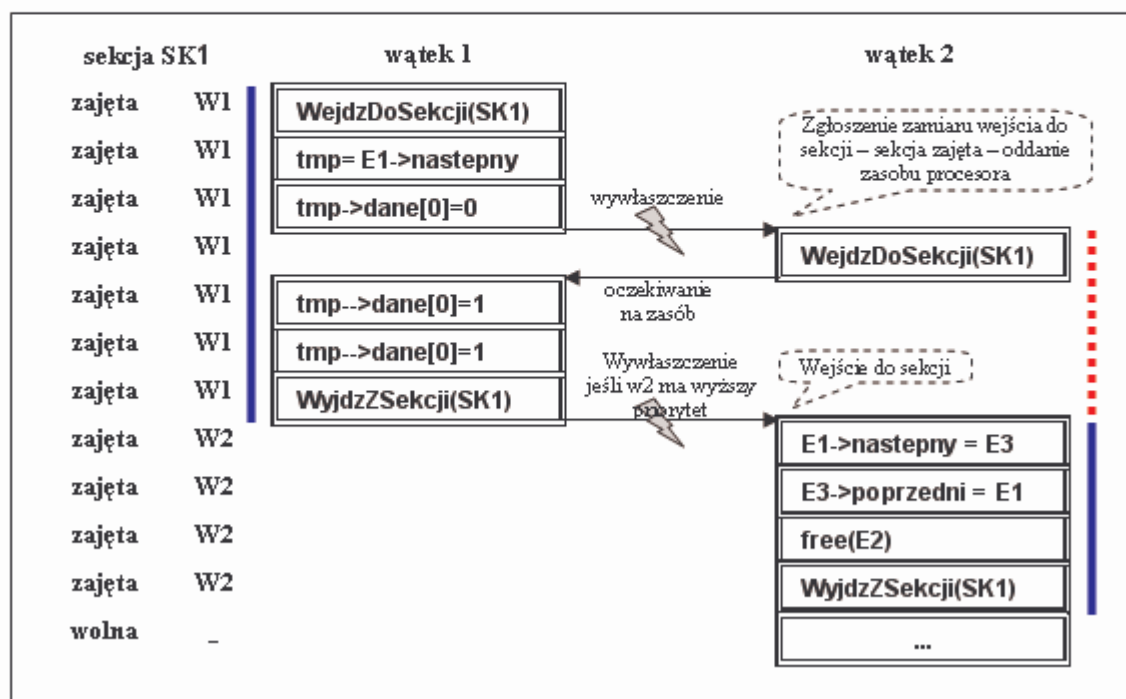
Sekcja krytyczna

- naraz w sekcji może znaleźć się tylko jeden proces/wątek
- jeżeli jakiś wątek zajął sekcję krytyczną – pozostałe mogą czekać albo sprawdzić, czy sekcja jest dostępna

Podstawowe operacje:

- wejdź do sekcji
- spróbuj wejść do sekcji
- opuść sekcję

Sekcja krytyczna to obszar kodu (a nie danych) !!!!



3.3 Zdarzenia

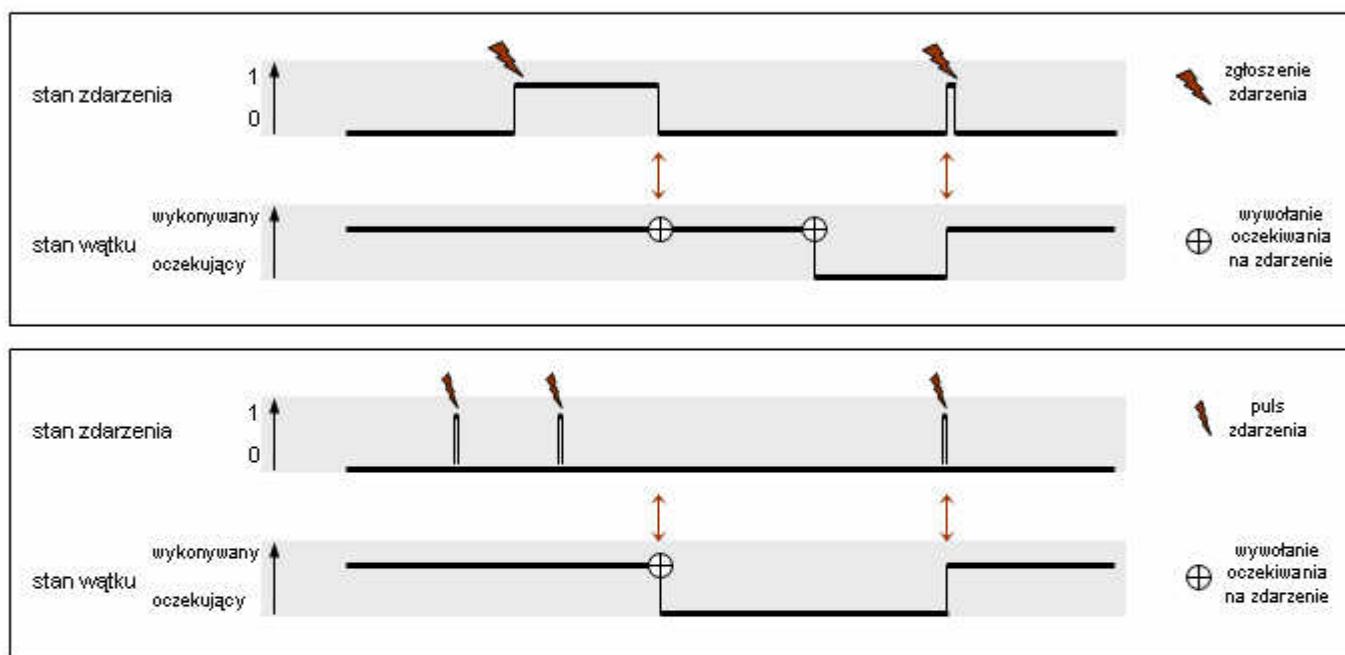
Zdarzenie posiada dwa stany:

- niezasygnalizowane (0),
- zasygnalizowane (1).

Podstawowe operacje:

- zgłoś zdarzenie
- zgłoś puls zdarzenia
- unieważnij zdarzenie
- oczekuj na zdarzenie

Oczekiwanie na niezaistniałe jeszcze zdarzenie zmienia stan procesu z wykonywanego na oczekujący. W momencie nadejścia zdarzenia proces oczekujący jest wstawiany do kolejki zadań gotowych do wykonania.



3.4 Semafor

Semafor:

- zliczające,
- binarne,
- mutex <ang. MUTual EXclusion> (właściciel!)

Zasięg: semafor lokalny i globalny.

Podstawowe funkcje na semaforze (atomowe -> wspomagane sprzętowo):

- **czekanie:**

```
Czekaj( S ):
    while( S ≤ 0 )
        Oczekiwanie;
    S = S - 1;
```

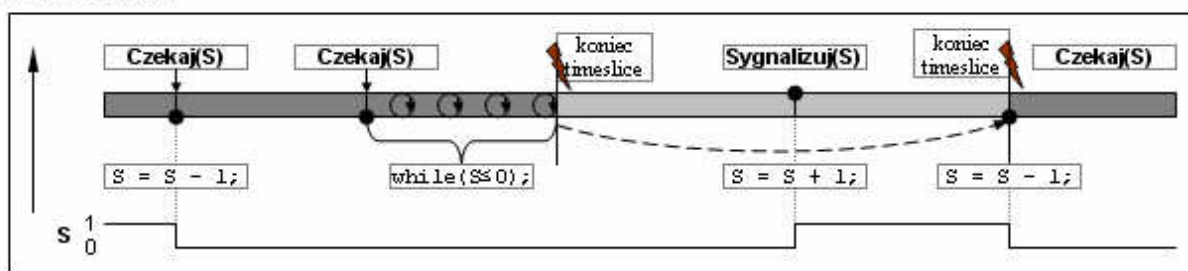
- **sygnalizowanie:**

```
Sygnalizuj( S ):
    S = S + 1;
```

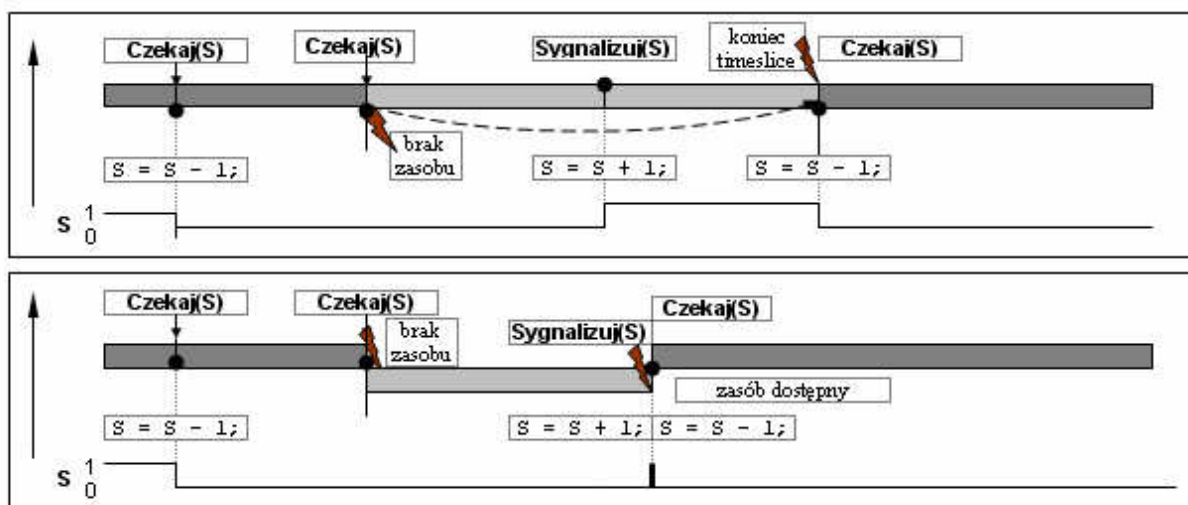
Oczekiwanie na semafor:

- **aktywne** – wirująca blokada – stosowane w systemach wieloprocessorowych (przy krótkich czasach oczekiwania),
- **pasywne** – powszechnie stosowane – usunięcie wątku z kolejki wątków gotowych do wykonania

Oczekiwanie aktywne



Oczekiwanie pasywne



Semafor ze zbiorem zadań oczekujących: sygnalizowanie wznowia wykonywanie pewnego (nieokreślonego) procesu. Semafor z kolejką zadań oczekujących: sygnalizowanie wznowia wykonanie określonego procesu z kolejki oczekujących na danym semaforze – najczęściej kolejność FIFO.

Semafor z listą procesów:

```
struct semaforFIFO{
    int wartosc;
    struct watek *pierwszy_oczekujacy;
}
```

- czekanie:

```
Czekaj( S ):
    S.wartosc --;
    if ( S.wartosc < 0 )
    {
        Dolacz_do_kolejki(S->pierwszy_oczekujacy);
        Przelacz_wykonanie();
    }
```

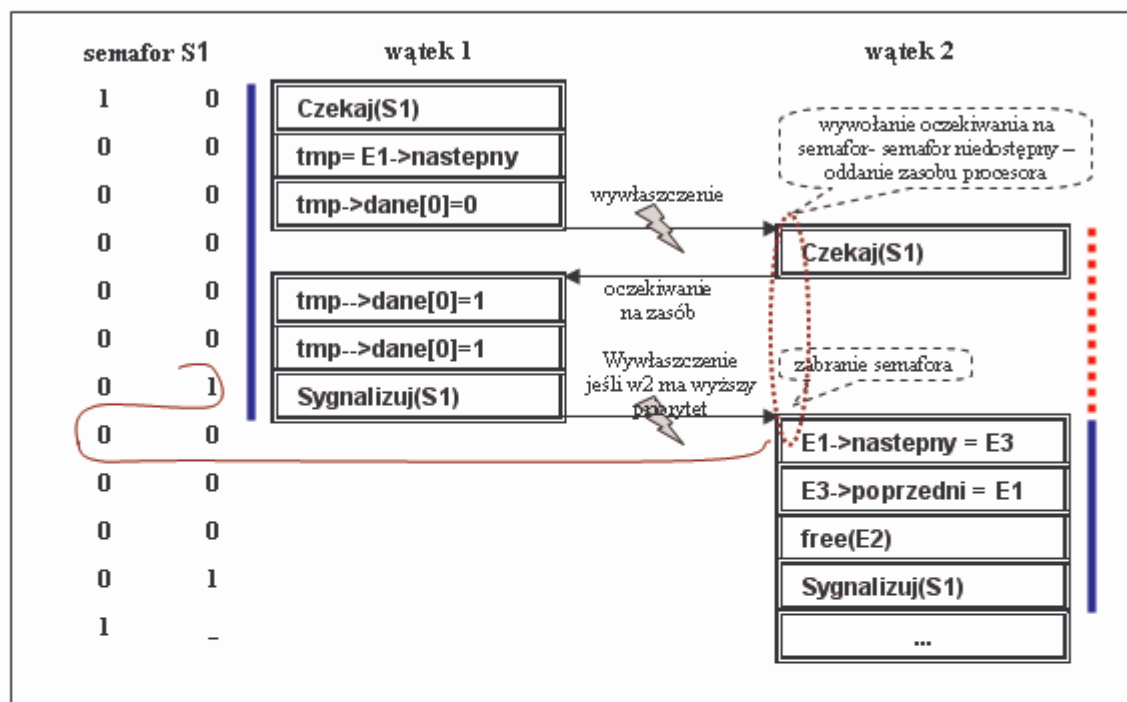
- sygnalizowanie:

```
Sygnalizuj( S ):
    S.wartosc = S.wartosc + 1;
    if( S.wartosc >= 0 )
    {
        Obudz( S->pierwszy); //atomowe!
        S->pierwszy = S->pierwszy->nastepny;
    }
```

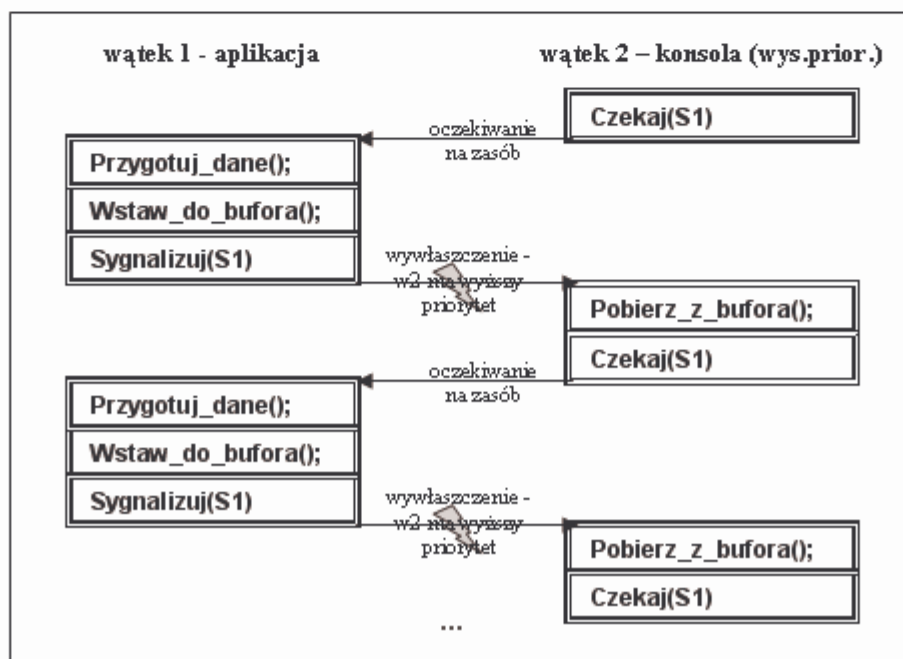
Silnie uczciwy semafor – jeśli nieskończenie wiele razy przyjmie wartość dodatnią, to każdy proces oczekujący na nim kiedyś wykona się.

Słabo uczciwy semafor – jeśli nieskończenie długo będzie przyjmował wartość dodatnią, to każdy proces oczekujący na nim kiedyś wykona się.

Implementacja sekcji krytycznej przy pomocy semafora

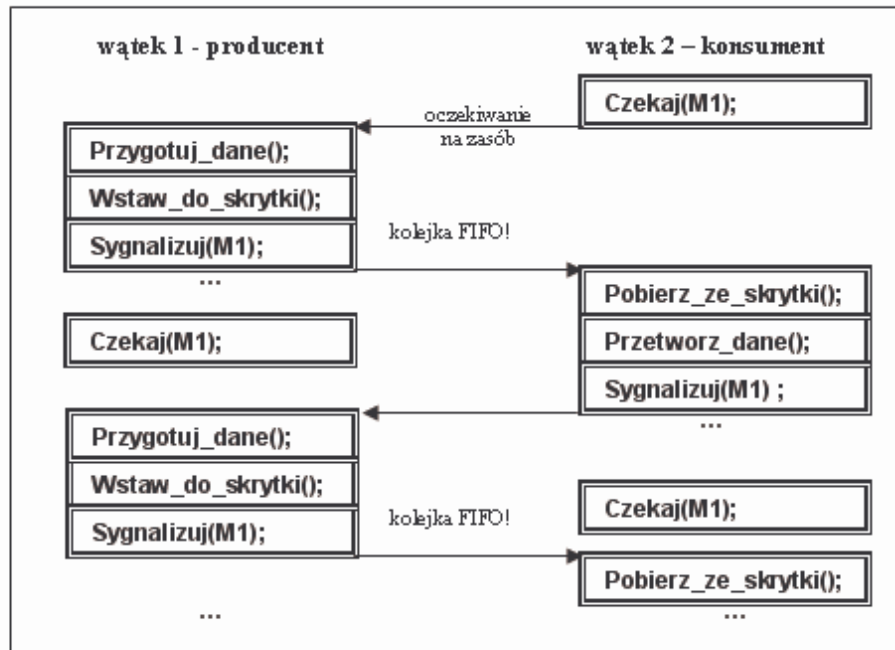


Synchronizacja dwóch procesów przy pomocy semafora



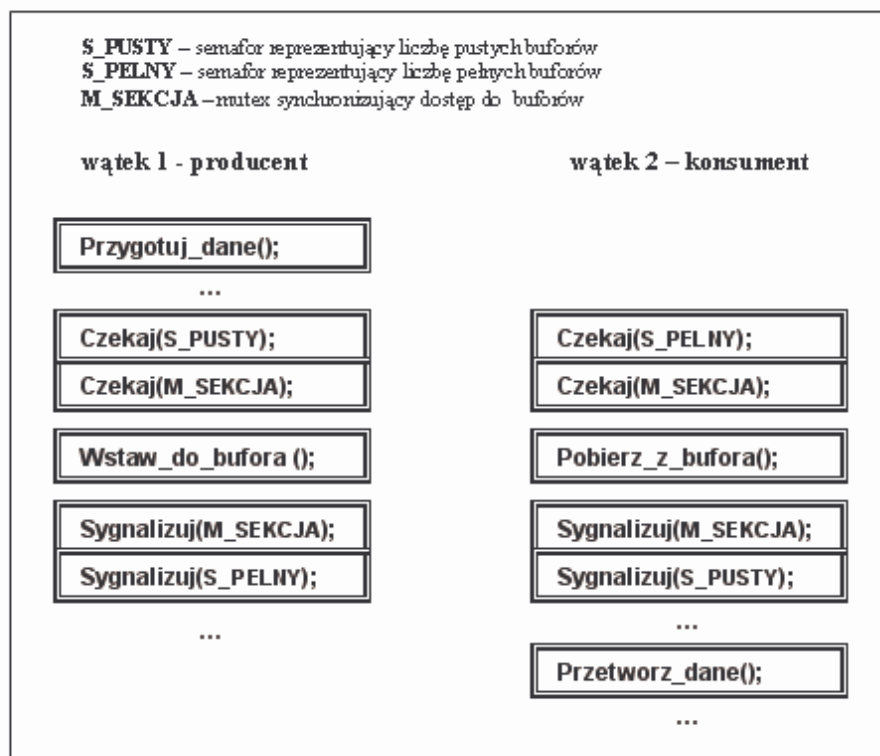
3.5 Klasyczne problemy synchronizacji procesów

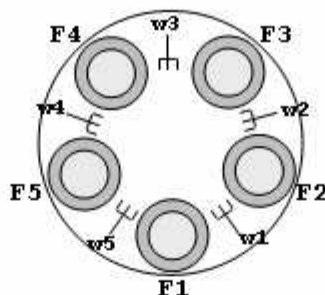
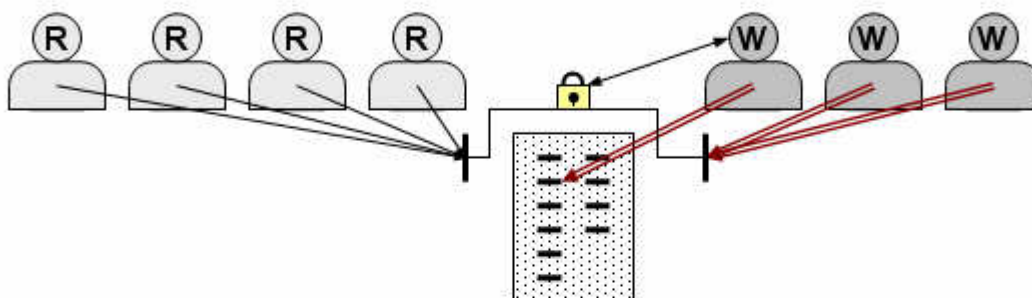
Model producent-konsument (jednostka danych)



Model producent – konsument z buforem ograniczonym

Oprócz semafora mutex, synchronizującego wielodostęp do buforów stosowane są semafory zliczające, wyznaczające liczbę pustych i pełnych buforów [1].



Problem 5 filozofów**Problem czytelników i pisarzy****3.6 Transakcje**

Transakcja – grupa operacji na danych, z zewnątrz widoczna jako jedna operacja.

Cechy transakcji (ACID):

- Atomic – atomowa – widziana z zewnątrz jako jedna operacja atomowa na danych
- Consistent – spójna – wykonanie transakcji nie powoduje utraty spójności danych
- Isolated – izolowana – działanie transakcji jest izolowane od innych
- Durable – trwała – zmiany pomyślnego zakończenia transakcji są stałe

2PC – 2 Phase Commit – dwufazowy protokół zatwierdzania

- faza pierwsza – wysłanie zamówień na realizację składowych transakcji
- faza druga:
 - zatwierdzenie transakcji (*commit*), jeśli wszystkie zamówienia mają potwierdzenie gotowości do realizacji
 - wycofanie transakcji (*rollback*), jeśli co najmniej jedna składowa nie jest dostępna.

Operacje wewnątrz transakcji muszą wspierać model transakcyjny !!!!

```
X = 1;
Y = 4;
```

Zacznij transakcję

- PrzypiszTransakcyjnie (X , 2);
- PrzypiszTransakcyjnie (Y , 3);

Zatwierdź transakcję

```
(X = 2, Y = 3)
```

```
X = 1;
Y = 4;
```

Zacznij transakcję

- PrzypiszTransakcyjnie (X , 2);

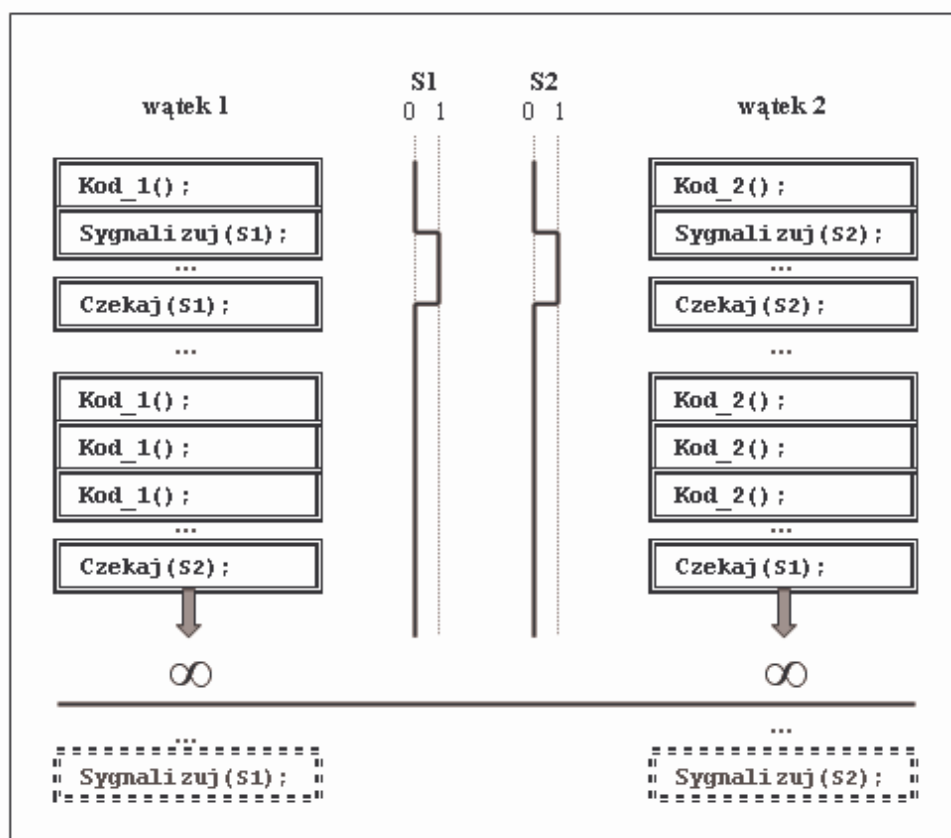
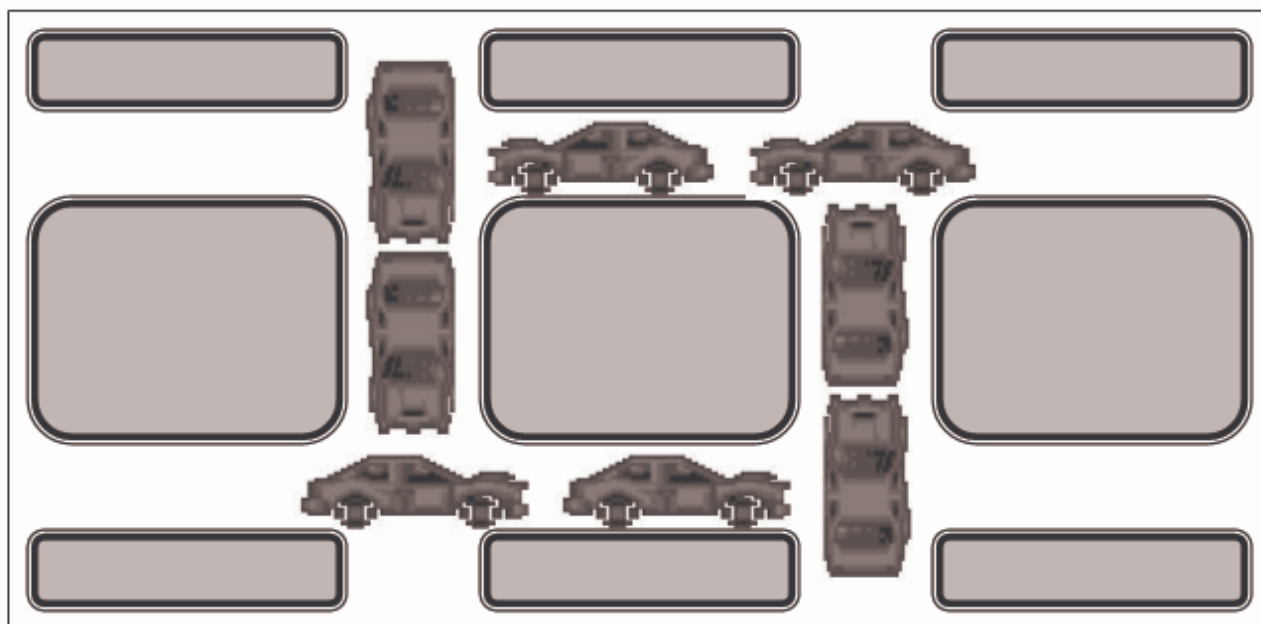
Wycofaj transakcję

```
(X = 1, Y = 4)
```

3.7 Zakleszczenia

Zakleszczenie – stan zbioru dwóch lub więcej wątków, powstały wskutek wzajemnego zawłaszczania wspólnych zasobów, objawiający się sytuacją, w której żaden z wątków tej grupy nie może dalej się wykonywać.
(każdy wątek oczekuje na zasób zawłaszczony przez inny wątek z grupy)

zakleszczenie - blokada - impas



Warunki powstawania zakleszczeń [1]

- wzajemne wykluczanie - co najmniej jeden zasób, do którego jednocześnie może mieć dostęp tylko jeden wątek
- przetrzymywanie i oczekiwanie – musi zaistnieć sytuacja, w której jakiś wątek zawłaszczył zasób i czeka na inny
- brak wywłaszczeń – tylko wątek może zwolnić zawłaszczony przez siebie zasób
- czekanie cykliczne – musi istnieć zbiór procesów znajdujących się w stanie przetrzymywania i oczekiwania

Graf przydziału zasobów [1]

$P = \{P_1, P_2, \dots, P_n\}$ – zbiór procesów w systemie

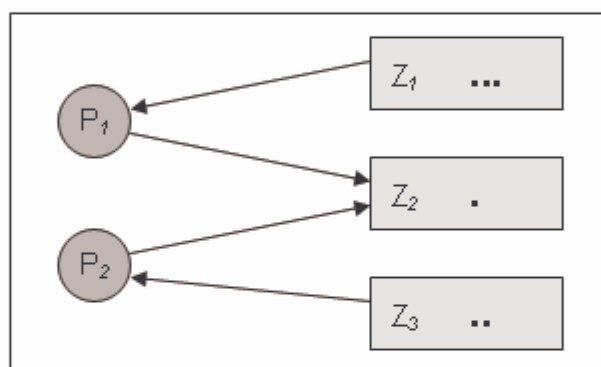
$Z = \{Z_1, Z_2, \dots, Z_m\}$ – zbiór zasobów w systemie

$K = \{P_1 \rightarrow Z_2, Z_2 \rightarrow P_1, P_1 \rightarrow Z_3, P_2 \rightarrow Z_2, \dots\}$ – zbiór krawędzi grafu, reprezentujący relacje między procesami a zasobami

$P_i \rightarrow Z_j$ - proces i żąda zasobu j

$Z_a \rightarrow P_b$ - zasób a został przydzielony procesowi b

Zasób może mieć jeden lub wiele egzemplarzy.

**Wykrywanie wystąpienia zakleszczenia**

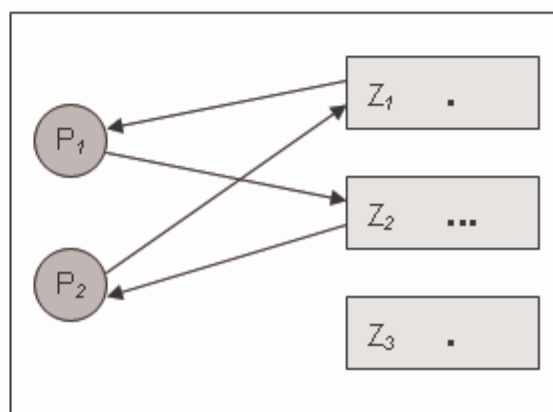
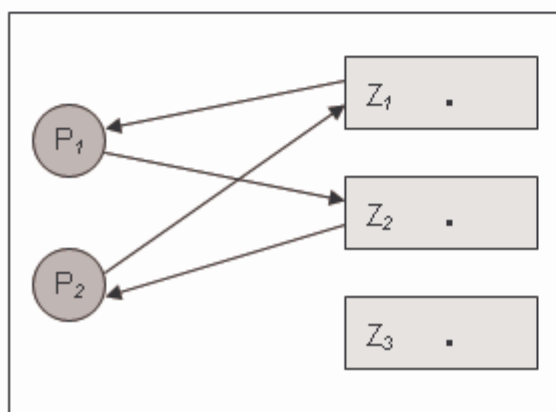
Jeżeli wszystkie zasoby mają jeden egzemplarz –

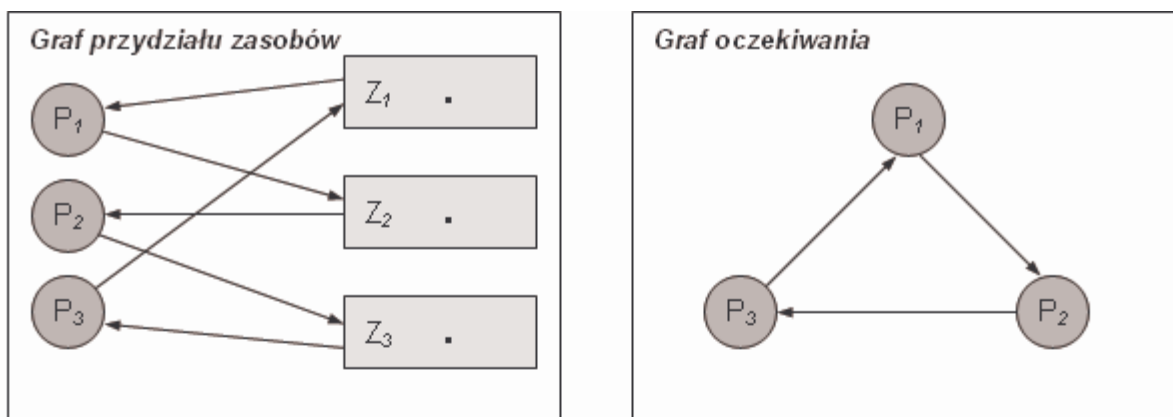
wystąpienie cyklu w grafie jest warunkiem koniecznym i wystarczającym zakleszczenia.

- każdy proces, który jest objęty cyklem, uczestniczy w zakleszczeniu.

Jeżeli istnieją zasoby, które mają wiele egzemplarzy –

wystąpienie cyklu w grafie jest warunkiem koniecznym zakleszczenia.





Postępowanie z zakleszczeniami w S.O.

- unikanie/zapobieganie zakleszczeniom,
- usuwanie zakleszczeń – może wiązać się z utratą danych,
- lekceważenie zakleszczeń -> zrzucanie obowiązku kontroli sytuacji na twórców oprogramowania.

Zapobieganie zakleszczeniom

Zapobieganie zakleszczeniom -> eliminacja warunków koniecznych do powstania zakleszczenia

- wzajemne wykluczanie - część zasobów nie może być współdzielona – wówczas warunku nie da się osłabić,
- przetrzymywanie i oczekiwanie - przykładowe rozwiązania:
 - o rezerwacja wszystkich potrzebnych zasobów na początku działania procesu,
 - o zamawianie tylko jednego zasobu naraz
- czekanie cykliczne – np.: numeracja zasobów i przydział tylko wg rosnących numerów -

Unikanie zakleszczeń

Unikanie zakleszczeń -> zarządzanie zasobami przez system operacyjny: rezerwacja, przydział

Stan bezpieczny (zasoby wieloegzemplarzowe)

Stan bezpieczny – istnieje porządek, w którym S.O. może przydzielić zasoby procesom.

Ciąg bezpieczny procesów : $(P_1, P_2, \dots, P_i, P_{i+1}, \dots, P_n)$ - system może tak przydzielać zasoby, aby wykonanie procesu P_{i+1} było możliwe po zakończeniu się procesu P_i .

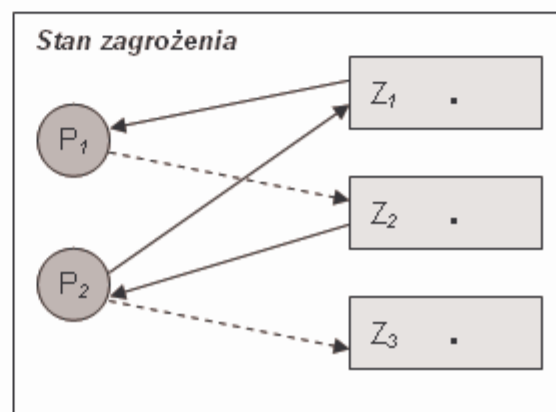
Graf przydziału zasobów z deklaracjami (zasoby jednoegzemplarzowe)

Powoływany proces składa deklaracje zamawianych zasobów

Przydział zasobu Z₂ procesowi P₁ jest możliwy tylko

(zamiana krawędzi deklaracji <P₁,Z₂> na krawędź przydziału)

Istnienie cyklu w grafie (oba rodzaje krawędzi) to stan zagrożenia.



3.8 Sieci Petriego

Sieci Petriego służą do modelowania i analizy systemów współbieżnych. Sieć Petriego jest czwórką (P, T, A, M_0) , gdzie:

P – zbiór miejsc,

T – zbiór przejść (tranzycji),

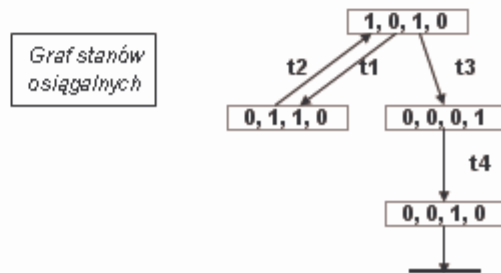
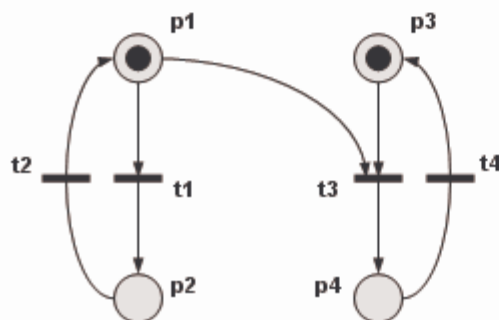
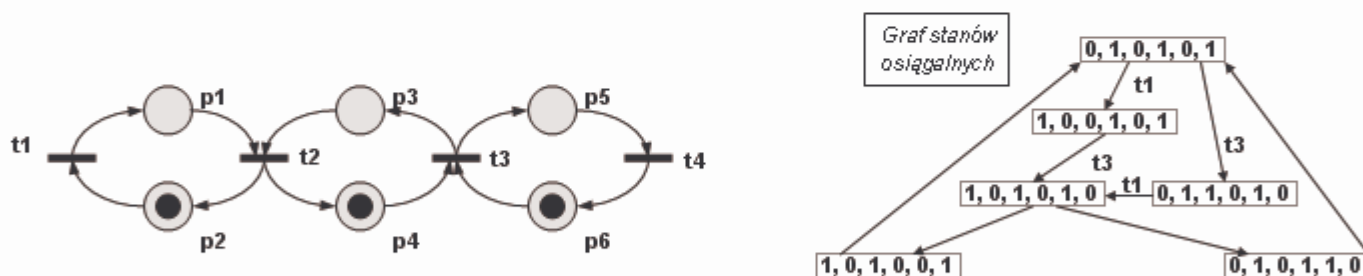
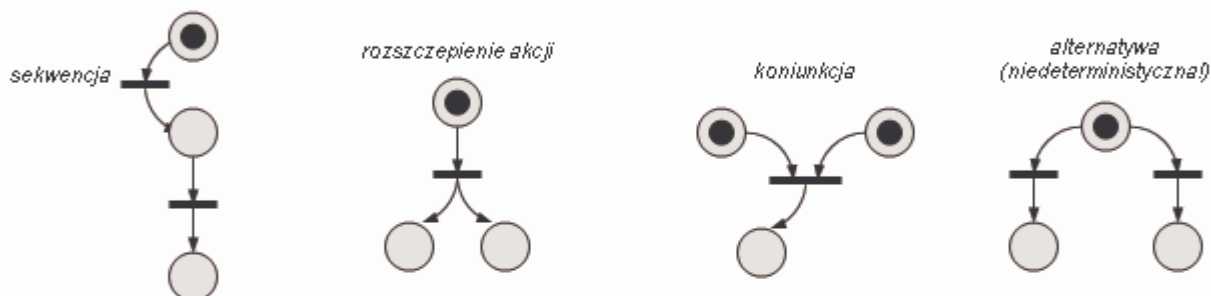
A – zbiór łuków,

M_0 – znakowanie początkowe.



Własności:

- łuki łączą tylko miejsca i przejścia,
- liczba znaczników dla dowolnego miejsca należy do N (0,1,2,3...),
- własności dynamiczne obrazuje przemieszczanie się znaczników w sieci,
- przejście jest wzbudzone i może zostać odpalone tylko wówczas, gdy wszystkie miejsca poprzedzające przejście są wzbudzone,
- przełączenie polega na zabraniu po 1 znaczniku z miejsc poprzedzających przejście i dodaniu po 1 do miejsc następujących.



3.9 Algorytmy wzajemnego wykluczania (mutual exclusion)

Rozwiązanie problemu wzajemnego wykluczania:

- zapobieganie blokadom,
- zapobieganie zagłodzeniom.

Założenia:

- zapis i odczyt wspólnych zmiennych jest atomowy
- jednocześnie operacje dostępu wykonywane sekwencyjnie
(kolejność nie jest znana)

Algorytm Dekkera [2 procesy]

K – wyznacza, czyja kolej, inicjacja 1 (lub 2)

S1 – oznacza żądanie wejścia procesu 1 do sekcji krytycznej (wartość 1), inicjacja = 0

S2 – oznacza żądanie wejścia procesu 2 do sekcji krytycznej (wartość 1), inicjacja = 0

Proces 1

```

funkcja_p1()
{
    while(1)
    {
        // poza sekcją krytyczną (sekcja lokalna)

        // zamiar wejścia do sekcji krytycznej
        S1 = 1;

        while( S2 == 1 )
        {
            if( K == 2 )
            {
                S1 = 0;
                while( K == 2 )    // ! aktywne oczekiwanie !
                    ;
                S1 = 1;
            }
        }

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        S1 = 0;
        K = 2;

        // poza sekcją krytyczną (sekcja lokalna)
    }
}

```

Proces 2

```

funkcja_p2()
{
    while(1)
    {
        // poza sekcją krytyczną (sekcja lokalna)

        // zamiar wejścia do sekcji krytycznej
        S2 = 1;

        while( S1 == 1 )
        {
            if( K == 1 )
            {
                S2 = 0;
                while( K == 1 )    // ! aktywne oczekiwanie !
                    ;
                S2 = 1;
            }
        }

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        S2 = 0;
        K = 1;

        // poza sekcją krytyczną (sekcja lokalna)
    }
}

```

Algorytm Dijkstry [N procesów]

stan[N] – tablica stanów procesów:

0 – sekcja lokalna

1 – zamiar wejścia do sekcji krytycznej

2 – sekcja krytyczna

kolej – pierwszeństwo wejścia do sekcji

! jeśli są procesy mające wyższy priorytet,
będą stale przejmować dostęp do sekcji krytycznej,
doprowadzając inne procesy do zagłodzenia

Proces i

```

funkcja_procesu( i )
{
    while(1)
    {
        stan[i] = 1;
        inny = kolej;
        while( inny != i )
        {
            if( stan[inny] == 0 )
                kolej = i;
            inny = kolej;
        }
        stan[i] = 2;
        wzajemowzajemstwo = 0;
        for( k=0; k < N; k++)
            if( k != i )
                if( stan[k] == 2 )
                    wzajemowzajemstwo = 1;
        if( wzajemowzajemstwo == 1 )
            continue;

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        stan[i] = 0;
    }
}

```

Algorytm Lamporta (piekarniany) [2 procesy]

Numer1, Numer2 – kolejność procesów

Proces 1

```

funkcja_p1()
{
    while(1)
    {
        // kod poza sekcją krytyczną

        // zamiar wejścia do sekcji

        Numer1 = 1;
        Numer1 = Numer2 + 1;

        while( Numer2 != 0 && Numer1 > Numer2 )
            ;

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        Numer1 = 0;
        // kod poza sekcją krytyczną
    }
}

```

Proces 2

```

funkcja_p2()
{
    while(1)
    {
        // kod poza sekcją krytyczną

        // zamiar wejścia do sekcji

        Numer2 = 1;
        Numer2 = Numer1 + 1;

        while( Numer1 != 0 && Numer2 > Numer1 )
            ;

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        Numer2 = 0;
        // kod poza sekcją krytyczną
    }
}

```

Algorytm Lamporta (piekarniany) [N procesów]

wybijający[N] – proces wybiera numer
 numer [N] – numer w kolejce do wejścia

! problem wyczerpania zakresu numerów
 (jeśli sekcja jest stale zajęta)

Proces i

```

funkcja_procesu( i )
{
    while(1)
    {
        //sekcja lokalna
        wybierający[i] = 1;
        numer[i] = max(numer) + 1;
        wybierający[i] = 0;

        for(j=0;j<N;j++)
        {
            if( j != i )
            {
                while( wybierający[j] != 0 );
                while( numer[j] != 0
                    && ( numer[i] > numer[j] ||
                        ( numer[i] == numer[j] && i > j ) ) );
            }
        }

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        numer[i] = 0;
    }
}

```

Algorytm Petersona [2 procesy]

K1, K2 – zmienne oznaczające zamiar wejścia do sekcji = {0,1}, inicjacja = 0;

Ustepujacy – który chce ustąpić przy współzawodnictwie = {1,2}, inicjacja np.=1.

Proces 1

```
funkcja_p1()
{
    while(1)
    {
        // kod poza sekcją krytyczną

        // zamiar wejścia do sekcji
        K1 = 1;
        Ustepujacy = 1;

        while( K2 != 0 && Ustepujacy == 1 )
            ;

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        K1 = 0;
        // kod poza sekcją krytyczną
    }
}
```

Proces 2

```
funkcja_p2()
{
    while(1)
    {
        // kod poza sekcją krytyczną

        // zamiar wejścia do sekcji
        K2 = 1;
        Ustepujacy = 2;

        while( K1 != 0 && Ustepujacy == 2 )
            ;

        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        K2 = 0;
        // kod poza sekcją krytyczną
    }
}
```

Algorytm Petersona [N procesów]

Tablice współdzielone:

stan[N] – stan procesu

kolej[N] – wybierana kolejność wykonania procesu

zmienne lokalne procesu:

inny -

który -

j, k - zmienne pomocnicze

Proces i

```
funkcja_procesu( i )
{
    while(1)
    {
        // sekcja lokalna
        for(k=0; k<= n-1 -1; k++) // tablica C
        {
            stan[i]=k;
            kolej[k]=i;
            do{
                if( kolej[k] == i )
                    break;
                for ( j=0; j<n-1; j++ )
                {
                    if(j != i)
                        inny = stan[j];
                    if( inny >=k )
                        break;
                }
            } while( inny < K );
        }

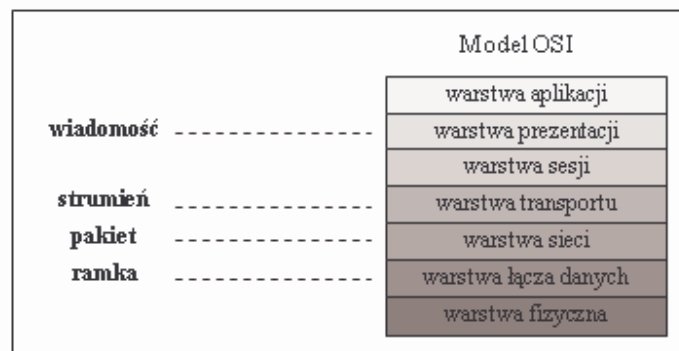
        // początek sekcji krytycznej
        OperacjeKrytyczne();
        // koniec sekcji krytycznej

        stan[i]:=0;
    }
}
```

4.1 Model warstwowy OSI

Wzorcowy model połączeń w systemach otwartych (ang. Open Systems Interconnection Reference Model (OSI)), przygotowany przez Biuro Międzynarodowych Standardów (ISO), zakłada wyodrębnienie siedmiu warstw systemu:

- fizycznej,
- łącza danych,
- sieci,
- transportu,
- sesji,
- prezentacji,
- aplikacji.



Rys.4.1 Formy przesyłanych danych w kontekście do modelu OSI

Warstwa fizyczna obejmuje interfejsy elektryczne i mechaniczne oraz zagadnienia transmisji zer i jedynek w medium. Ich poprawne przesyłanie w postaci porcji, zwanych **ramkami**, jest zadaniem urządzeń sieciowych, działających w warstwie łącza danych. Warstwa sieciowa odpowiada za prawidłowe skierowanie paczki danych (**pakietu**) do adresata oraz odpowiednie wyznaczanie jej trasy. Warstwa transportowa jest odpowiedzialna za poprawne przekazywanie danych między jednostkami komunikującymi się. W szczególności dotyczy to niezawodnych połączeń transportowych, które przesyłają dane spójnym ciągiem, zachowującym kolejność bajtów, zwanym **strumieniem**. Obowiązkiem warstwy transportowej jest kontrola przesyłania pakietów (podział strumienia na pakiety), potwierdzanie dostarczeń, retransmisja, ochrona przed duplikacją, itp. Warstwa sesji jest odpowiedzialna za nawiązywanie komunikacji między jednostkami oraz kontrolę nad jej przebiegiem. Warstwa prezentacji wiąże się z interpretacją danych przekazywanych między maszynami. Odpowiedzialna też jest za ich konwersję w przypadku odmiennej ich reprezentacji na komunikujących się maszynach. Tutaj także wyodrębnia się logiczną, spójną porcję danych, będącą **wiadomością** (komunikatem). Ostatnią warstwę stanowią aplikacje końcowe, korzystające z zasobów systemu rozproszonego. Zestawienie warstw i wymienionych form przesyłanych danych obrazuje rysunek 4.1.

4.2 Modele wymiany danych

Między komputerami w sieci dokonywana jest wymiana danych. Za elementarną akcję przyjmuje się przekazanie komunikatu między procesami. Najczęściej spotykane są następujące relacje pomiędzy uczestnikami komunikacji: producent/konsument, master/slave i klient/serwer.

Model producent/konsument

Model producent/konsument jest najprostszym modelem wymiany danych. Polega on na wyróżnieniu jednej jednostki jako producenta danych. Druga jednostka (lub grupa jednostek) jest konsumentem. Rolą producenta jest wysyłanie wiadomości, których odbiorcą jest konsument. Model ten nie obejmuje żadnych dodatkowych zależności współdziałania między komputerami.

Model master/slave

Model ten polega na uczynieniu jednej jednostki (lub grupy) nadrzędnym (master) wobec pozostałych (slave). Rola stacji slave jest tu ograniczona do odpowiedzi na zapytania stacji master i wykonywania działań narzuconych przez jednostkę nadrzędną. Takie rozwiązanie znacznie ułatwia badanie zachowania determinizmu systemu, co jest ważne w projektowaniu i analizie systemów czasu rzeczywistego.

Model klient/serwer

Jest to najpopularniejszy model komunikacji pomiędzy komputerami. Zakłada on istnienie serwera, który świadczy usługi na rzecz klienta (lub grupy klientów). Usługą może być przekazanie czy zapisanie pewnych danych, usługa może mieć charakter obliczeniowy. Wykonanie akcji dzieli się na trzy etapy:

- wywołanie usługi serwera poprzez klienta,
- wykonanie usługi przez serwer,
- zwrócenie efektów (wyników) klientowi.

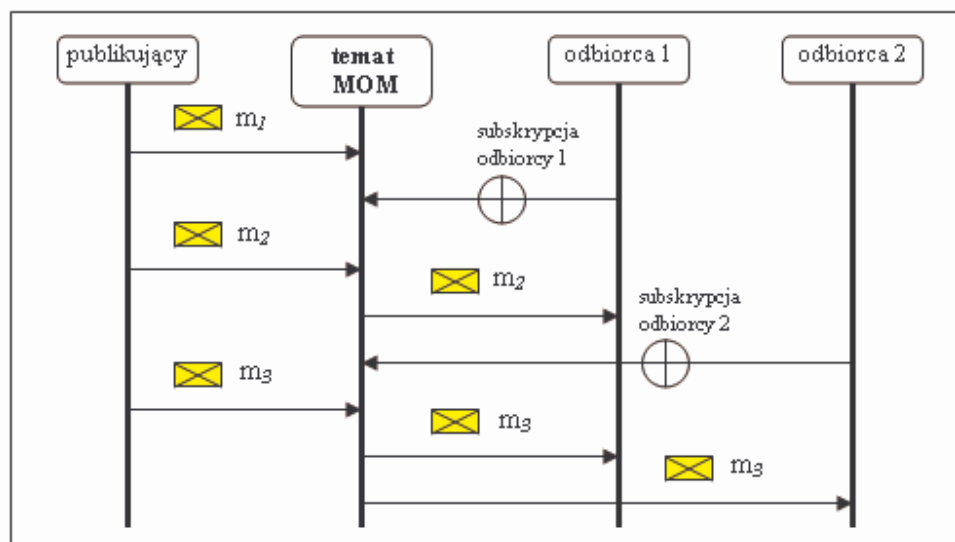
Serwer oczekuje na zamówienia ze strony klienta (klientów). W trakcie wykonywania usługi klient może być zablokowany oczekiwaniem na wyniki. Warto wspomnieć, że w przypadku dużego obciążenia serwera często stosuje się architekturę klastra. Serwer stanowi wówczas grupa jednostek obliczeniowych. Z punktu widzenia klienta nadal jest to jeden serwer.

4.3 Rozgłaszanie grupowe

Proste rozsyłanie danych do wielu odbiorców jest najczęściej realizowane przez rozgłaszanie grupowe (ang. multicast). Stosowane jest także rozsyłanie do wszystkich użytkowników sieci (ang. broadcast).

Rozgłaszanie grupowe w modelu zorientowanym na przesyłanie wiadomości (MOM) polega na wysyłaniu komunikatu do wielu odbiorców. Model ten jest często znany jako 'publikuj / subskrybuj'. Publikatorem jest jednostka nadająca wiadomość, subskrybentem (prenumeratorem) – odbierająca. W modelu tym nie następuje blokowanie podczas wysyłania komunikatu, nie ma też potwierdzeń. Rozgłaszanie grupowe najczęściej nie gwarantuje dostarczenia danych - wówczas w przypadkach wymagających potwierdzania wiadomości dekomponuje się system na poszczególne połączenia typu punkt-do-punktu.

Publikowanie wiadomości wiąże się z umieszczeniem jej w ogólnodostępnej kolejce, zwanej tematem (ang. topic). Komunikaty tam wstawione docierają do odbiorców, którzy wcześniej zapisali się do odbioru danych. Scenariusz działania rozsyłania grupowego do tematu przedstawia rys. 4.2.



Rys. 4.2. Scenariusz rozsyłania grupowego w MOM

4.4 Działania gwarantowane i niegwarantowane

Przy projektowaniu systemu wielozadaniowego zakłada się, że przesyłane między procesami wiadomości docierają do adresata. Gwarancję wówczas zapewnia system operacyjny. W systemach rozproszonych nie można zagwarantować dostarczenia danych, dlatego wprowadza się potwierdzenia, gdy jest to pożądane. Mechanizm ten, wraz z nadawaniem komunikatom etykiet, zapobiega także duplikacji wiadomości.

Działania niegwarantowane mogą być zadowalającą formą komunikacji w wielu systemach. Ma to miejsce np. podczas transmisji obrazu w czasie rzeczywistym (telekonferencje), gdzie dopuszcza się zagubienie pojedynczych ramek przesyłanego obrazu. Istnieją jednak systemy (np. finansowe), w których konieczne jest zapewnienie gwarantowanego przekazywania wiadomości bez ich duplikacji. Szczególnie dotyczy to sytuacji, gdy nie tyle jest ważna historia, co aktualność danych. Wówczas dodaje się do modelu wiadomości atrybut trwałości (*ang. persistency*). Dane z ustawionym atrybutem trwałości nie mogą zginąć z systemu – nawet po jego awarii proces naprawczy powinien umieć je odzyskać.

Celem zwiększenia niezawodności operacji wykonywanych podczas komunikacji można stosować dwufazowe zatwierdzanie, zwane **transakcją**. W pierwszej fazie wykonuje się rezerwację zasobów i sprawdza się możliwość wykonania operacji. Faza druga polega na zatwierdzeniu wszystkich operacji w obrębie transakcji. Pomyślny przebieg transakcji wiąże się z osiągnięciem planowanych wyników. W przypadku braku możliwości spełnienia choć jednej operacji w obrębie transakcji odwoływane są wszystkie pozostałe i przywracany jest stan początkowy.

4.4.1 Działania buforowane i niebuforowane

W modelach komunikacji pomiędzy dwoma komputerami (ew. ich większą liczbą) zakłada się najczęściej gotowość systemu do wysłania czy odebrania wiadomości. Może się zdarzyć, że w systemie pojawią się dwie lub więcej wiadomości oczekujące do nadania albo że serwer otrzyma kilka wiadomości nie będąc w stanie gotowości na ich przetworzenie. Wówczas rozwiązaniem problemu jest zabronienie wysyłania danych poprzez blokowanie procesu nadawcy albo gromadzenie zleceń w kolejkach komunikatów, zarówno po stronie nadawcy jak i odbiorcy. Takie gromadzenie zgłoszeń poprzez system operacyjny lub dedykowany do tej czynności nazywa się działaniem buforowanym. Pozostałe działania określa się mianem niebuforowanych.

4.5 Metody komunikacji

Komunikacja między dwoma jednostkami może odbywać się w sposób synchroniczny albo asynchroniczny. Podział ten przeprowadza się na podstawie przebiegu procesu dostarczenia komunikatu.

Komunikacja synchroniczna

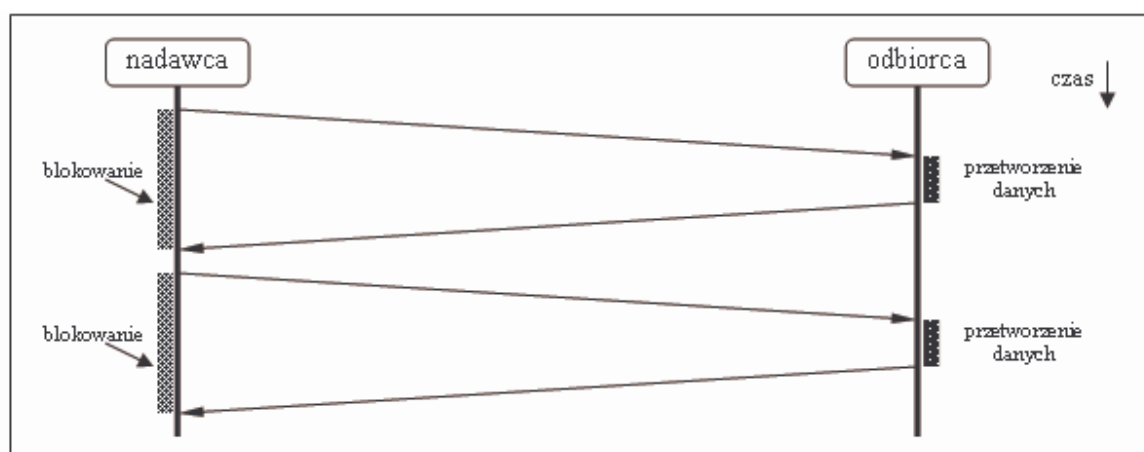
Komunikacja synchroniczna ma miejsce w przypadku, gdy wysłaniu wiadomości towarzyszy zablokowanie nadawcy do czasu otrzymania potwierdzenia. Rozwiązanie takie pozwala na łatwą kontrolę dostarczenia danych, możliwa jest również natychmiastowa interpretacja otrzymanego wyniku. Nie jest konieczna realizacja kolejowania wiadomości, zarówno po stronie nadawcy jak i odbiorcy. Dodatkowo zachowywana jest narzucona przez nadawcę kolejność przesyłania danych. Ewidentną wadą jest niska wydajność związana z przestojem na czas całej operacji. Komunikację synchroniczną jest łatwa w modelowaniu systemu, jego analizie poprawności i analizie czasowej. Rysunek 4.3 przedstawia diagram czasowy synchronicznej wymiany danych.

Komunikacja synchroniczna występuje w dwóch wariantach:

- zdalne wywołanie procedury,
- przekazywanie wiadomości.

Zdalne wywołanie procedury (*ang. Remote Procedure Call*) jest przeniesieniem wykonywania pewnej procedury na komputer odległy. Stosowanym modelem wymiany danych jest model klient – serwer. Serwer udostępnia klientom zdefiniowane odpowiednimi interfejsami usługi. Wywołanie procedury wiąże się z przekazaniem parametrów jej wywołania i zwróceniem wyników.

Synchroniczne przekazywanie wiadomości jest uboższą wersją zdalnego wywoływania procedury. Dane przekazywane są tylko w jedną stronę - od nadawcy do odbiorcy. W drugą stronę przesyłane jest tylko potwierdzenie odbioru lub zgłoszenie błędu. Nie ma jawnych wytycznych co do akcji, jakie wykonuje odbiorca z wiadomością.



Rys. 4.3 Diagram czasowy komunikacji synchronicznej

Komunikacja asynchroniczna

Komunikacja asynchroniczna różni się od synchronicznej tym, że nie blokuje nadawcy podczas jej trwania. Dostępne są również dwie jej formy:

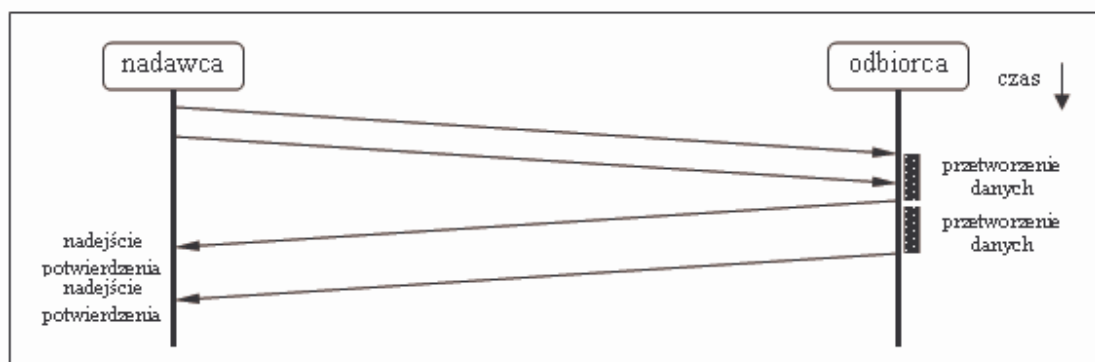
- asynchronicznie zdalne wywołanie metody,
- asynchroniczne przekazywanie wiadomości.

Asynchroniczne zdalne wywołanie procedury jest rozwiązaniem znacznie młodszym od wersji synchronicznej. Pomysł polega na rozdzieleniu wywołania procedury po stronie klienta na poziomie interfejsu na dwie fazy:

- wywołanie metody rozpoczynającej wykonanie procedury przez serwer,
- wywołanie metody odbierającej od serwera wyniki.

Metody te mają odpowiednio dobrane parametry – pierwsza ma tylko parametry wejściowe, druga – wyjściowe. Wywołanie pierwszej metody nie powoduje blokowania klienta – natychmiast zwracane jest sterowanie. Druga metoda, w zależności od implementacji systemu może być blokująca. Opcjonalne jest udostępnienie metody zwracającej przez serwer stan wykonania usługi.

Najczęstszą postacią wykorzystywania komunikacji asynchronicznej jest przekazywanie wiadomości. Ta prostsza forma komunikacji od zdalnego wykonania procedury polega na nadaniu komunikatu do odbiorcy bez blokowania oczekiwaniem na potwierdzenie (rys.4.4).



Rys. 4.4 Diagram czasowy komunikacji asynchronicznej

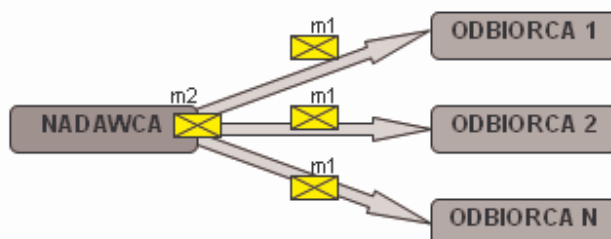
Potwierdzenie może nadejść po nieokreślonym czasie. Jest to niewątpliwą wadą, lecz znacznie poprawia wydajność systemu. Nadawca musi niestety osobiście sprawdzić status wysłanych danych. Komunikacja asynchroniczna wymaga implementacji kolejek wiadomości (p.4.5). W komunikacji synchronicznej gromadzenie się zamówień blokowane jest przez system. Spotyka się kompleksowe rozwiązania komunikacji asynchronicznej w postaci systemów wiadomości kolejkowanych.

4.6 Klasyfikacja krotności komunikacji

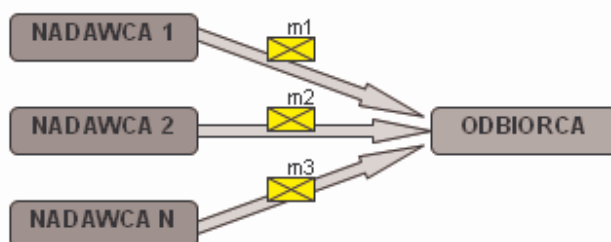
- Jeden do jednego (1:1) – pojedynczy proces może wysyłać tylko do innego (jednego) procesu, ów proces zaś może czytać dane tylko tego nadawcy, komunikacja może odbywać się jednostronnie lub obustronnie,



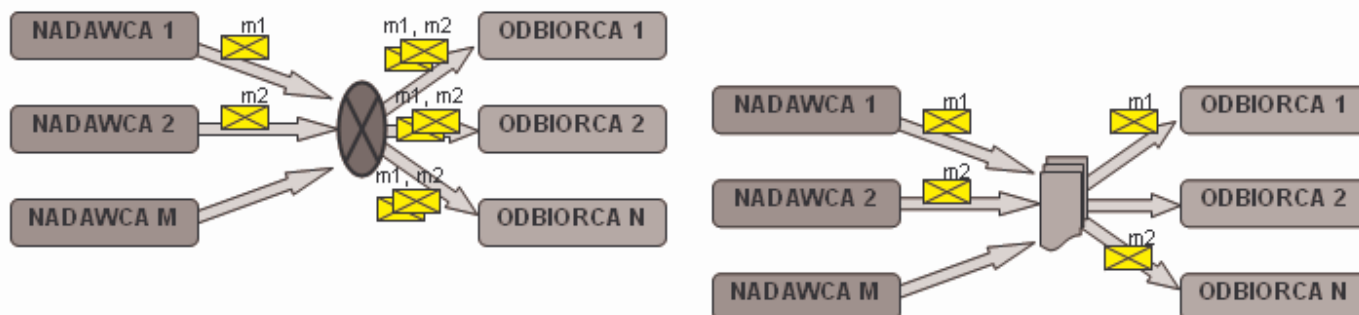
- Jeden do wielu (1:N) – pojedynczy proces może pisać do grupy procesów, każdy proces dostaje kopię wiadomości, często wraz z usunięciem tego procesu sesja komunikacji ulega zakończeniu, odbiorca może być subskrybentem lub adresatem



- Wiele do jednego (N:1) – wiele procesów może pisać do jednego procesu, który gromadzi unikalne wiadomości od każdego procesu (kolejka),



- Wiele do wielu (M:N) – każdy z pewnego zbioru procesów może wysyłać wiadomość do grupy, odczyt może odbywać się na dwa sposoby:
 - każdy proces otrzymuje pojedynczą kopię wiadomości, musi istnieć dystrybutor, odbiorca to np. subskrybent,
 - każda wiadomość występuje w jednym egzemplarzu – ściągana jest przez aktualnie czytający proces (kolejka).



Adresowanie:

- odbiorca anonimowy,
- odbiorca wskazany / grupa odbiorców,
- nadawca anonimowy,
- nadawca znany.

4.7 Mechanizmy komunikacji międzyprocesowej (IPC)

- pliki - rozwiązanie nieeleganckie ;)
- pamięć dzielona
- skrzynki (mailslot)
- potoki (pipe) – anonimowe i nazwane
- kolejki FIFO
- kolejki komunikatów <msg.h>
- kolejki komunikatów POSIX <mqueue.h>
- sygnały (UNIX)
- kolejka komunikatów systemowych (Win)
- zdalne (lokalne) wywołanie procedury RPC/LPC
- gniazda (Sockets)
- systemy wiadomości kolejkowanych (MQ)
- schowek (Clipboard)
- OLE/COM CORBA
- DDE

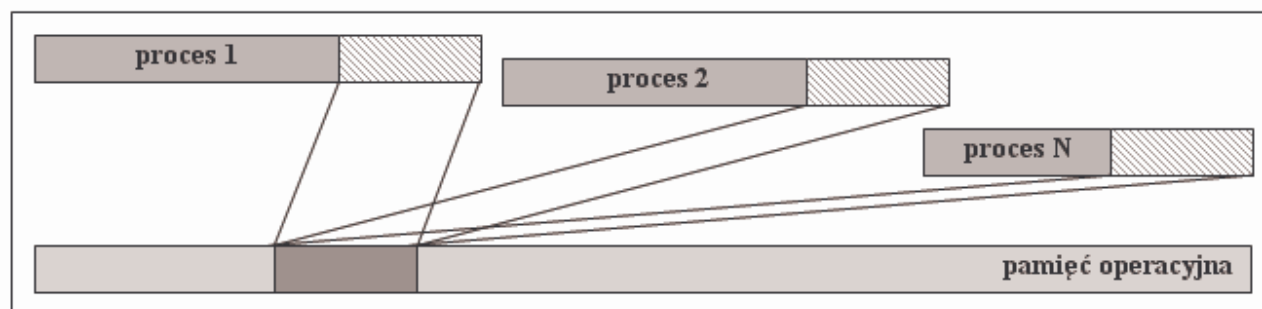
4.8 Pamięć dzielona

Pamięć dzielona to wspólny obszar adresowy dla różnych procesów.

- wskaźniki - adresacja różna dla różnych procesów,
- szybka wymiana danych, efektywne dla dużych bloków danych,
- brak synchronizacji dostępu !!!
- prawa dostępu: Linux - na zasadzie pliku, Windows – w oparciu o listy ACL,

Identyfikacja obiektu pamięci dzielonej:

- Linux: klucz (generowany w oparciu o plik i 8bitowy identyfikator – *ftok()*)
- Windows – nazwa (łańcuch tekstowy, np. "mojapamiec").



Rys. 4.5 Przykład działania pamięci dzielonej

Podstawowe operacje na pamięci dzielonej:

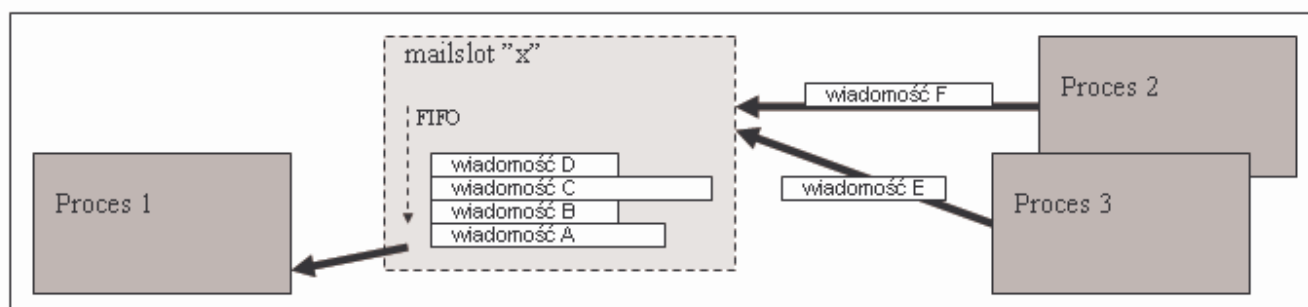
- tworzenie obiektu pamięci dzielonej,
- dołączanie/mapowanie obiektu pamięci dzielonej w procesach,
- operacje na pamięci dzielonej (kopiowanie, zmiana, operacje atomowe),
- odłączanie/odmapowanie obiektu pamięci dzielonej,
- niszczenie obiektu pamięci dzielonej.

UNIX / LINUX	Windows
- <code>shmget()</code> ;	- <code>CreateFileMapping()</code> ;
- <code>shmctl()</code> ;	- <code>OpenFileMapping()</code> ;
- <code>shmat()</code> ;	- <code>MapViewOfFile()</code> ;
- <code>shmdt()</code> ;	- <code>CopyMemory()</code> ; <code>memcpy()</code> ; <code>strcpy()</code> ;
- <code>memcpy()</code> ; <code>strcpy()</code> ;	- <code>UnmapViewOfFile()</code> ;
	- <code>CloseHandle()</code> ;

4.9 Skrzynki (ang. mailslot)

Mailslot – mechanizm komunikacji między procesami w obrębie jednego systemu lub w sieci, zorientowany na wiadomości.

- komunikacja jednokierunkowa,
- nie gwarantuje dostarczenia wiadomości, ochrony przed duplikacją ani zachowania kolejności wiadomości,
- tylko proces tworzący skrzynkę (lub jego proces potomny) może odczytywać jej zawartość w obrębie instancji s.o.,
- w sieci – odbierać wiadomości może jedna instancja na dany system operacyjny,
- dowolna liczba nadawców,
- kolejność wiadomości – FIFO,
- ograniczenie rozmiaru wiadomości – 64KB,
- ograniczenie rozmiaru wiadomości w sieci (broadcast, datagram) – 424B,
- dane przechowywane są w pamięci jądra s.o., nie są zapisywane (choć dostęp ma charakter systemu plików).



Rys. 4.6 Przykład działania skrzynek

Nazwy skrzynek:

\\.\mailslot\nazwa1	(lokalny)
\\.\mailslot\katalog1\katalog2\nazwa2	(lokalny, z katalogami)
\\komputer\mailslot\nazwa2	(adresowany do konkretnego węzła)
\\domena\mailslot\nazwa2	(adresowany do węzłów w zadanej domenie)
*\mailslot\nazwa2	(adresowany do węzłów w bieżącej domenie)

Windows
- CreateMailslot();
- GetMailslotInfo();
- SetMailslotInfo();
- CreateFile();
- WriteFile();
- ReadFile();
- CloseHandle();

4.10 Potoki (ang. pipe)

Potok anonimowy

Potok anonimowy służy do jednokierunkowej komunikacji strumieniowej między wątkami danego procesu lub między wątkami procesów macierzystego i potomnego. Dostęp do potoku odbywa się przez wskaźniki odczytu i zapisu, które są mają sens tylko w danej przestrzeni adresowej. Wskaźniki te można przekazać do wątków lub do procesów potomnych.

- dane są przesyłane w formie strumienia bajtów, bez wyróżniania wiadomości / kolejnych porcji,
- zapis do potoku anonimowego nie posiada duplexu: kto zapisał dane, ten może je odczytać (~kolejka),
- odczyt blokowany lub nieblokowany – blokowany: wywołanie funkcji czytającej zablokuje, jeśli nie ma danych,
- odczyt – obejmuje tyle danych, ile przekazano do odczytu lub tyle, ile było dostępnych (jeśli przekazano więcej),
- zapis do zapełnionego potoku jest blokowany do momentu odebrania danych, rozmiar bufora -> CreatePipe(),

- zapis – jeśli porcja danych zmieści się w buforze – zapis dokonuje się, jeśli nie, to blokuje zapis lub zapisuje część (n-b),
- kolejność bajtów danych jest zachowana (dla każdej operacji zapisu),
- kolejność wpisów – nie jest kontrolowana,
- potoki są usuwane albo na żądanie aplikacji, albo po zakończeniu ostatniego korzystającego z nich procesu.

Scenariusz użycia: proces macierzysty zostawia wskaźnik do zapisu, proces potomny do odczytu – albo odwrotnie.

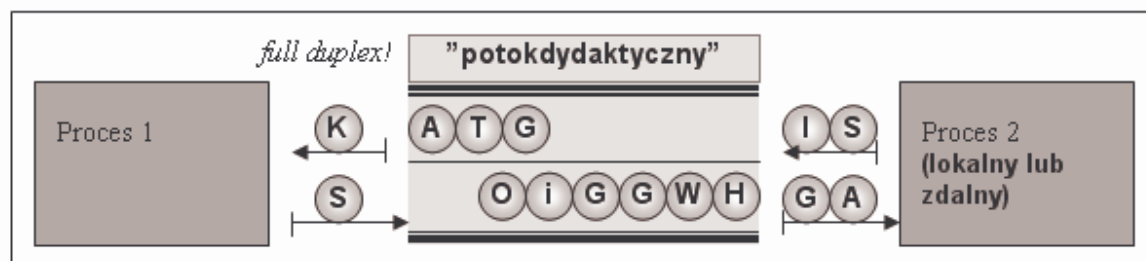


Rys. 4.7 Potoki anonimowe jednokierunkowe - przykład

LINUX <unistd.h>	Windows <windows.h>
- pipe();	- CreatePipe();
- read();	- WriteFile();
- write();	- ReadFile();
- close();	- CloseHandle();

Potok nazwany

Potok nazwany służy do pełnej dwukierunkowej (*full duplex*) komunikacji strumieniowej między dwoma wątkami spokrewnionych lub niespokrewnionych procesów w obrębie jednej instancji systemu operacyjnego lub w sieci. Implementacja: tylko Windows.



Rys. 4.8 Potoki nazwane - przykład

- zapewnia zachowanie kolejności bajtów, brak duplikatów i utrat danych,
- podstawowy model – strumieniowy, dostępny jest także model strumieniowy zorientowany na wiadomości,
- identyfikacja – przez nazwę (łańcuch tekstowy),
- serwer – proces tworzący i obsługujący potok, klient – proces dołączający się do potoku,
- jeśli serwer zamknie potok – klient nie dostanie informacji o tym, najbliższa operacja zakończy się błędem,
- jest możliwe tworzenie przez serwer wielu instancji jednego potoku dla wielu klientów,
- instancja s.o. pozwala na utworzenie tylko jednego serwera o danej nazwie (tylko jednemu procesowi),
- operacje odczytu/zapisu – synchroniczne lub asynchroniczne,
- specjalna opcja dla trybu strumieniowego przy połączeniach sieciowych – blokowanie operacji wpisu aż do odbioru przez potok na komputerze zdalnym (alternatywnie: kumulacje mniejszych porcji danych lub oczekiwanie przez pewien czas),

Nazwy potoków:

\\.\pipe\potokdydaktyczny

(potok nazwany, dostępny w bieżącej instancji systemu)

\\KomputerDocelowy\pipe\potokdydaktyczny

(potok nazwany, dostępny na komputerze KomputerDocelowy)

Windows
- <code>CallNamedPipe()</code> ;
- <code>ConnectNamedPipe()</code> ;
- <code>CreateNamedPipe()</code> ;
- <code>DisconnectNamedPipe()</code> ;
- <code>GetNamedPipeHandleState()</code> ;
- <code>GetNamedPipeInfo()</code> ;
- <code>PeekNamedPipe()</code> ;
- <code>SetNamedPipeHandleState()</code> ;
- <code>TransactNamedPipe()</code> ;
- <code>WaitNamedPipe()</code> ;

4.11 Kolejki FIFO

Kolejki FIFO umożliwiają komunikację między procesami niespokrewnionymi ze sobą na zasadzie klient-serwer. Komunikacja jest obustronna (jest to odpowiednik potoków nazwanych).

- możliwe jest wystąpienie większej ilości wpisujących i odczytujących,
- rolę nadawcy/odbiorcy (pisarza/czytelnika) ustala się podczas dołączania do kolejki,
- adresacja odbywa się poprzez podanie ścieżki do obiektu w systemie plików,
- dane są formowane w wiadomości, wiadomości nie posiadają adresata,
- kolejność wstawianych danych jest zachowana (FIFO),
- każdy czytelnik indywidualnie odczytuje wiadomość z początku kolejki, jednocześnie usuwając ją,
- spójność danych jest gwarantowana, o ile rozmiar danych nie przekracza rozmiaru bloku (4096)
- dane przekraczające rozmiar 4096B dzielone są na części, które mogą być 'przeplatane' z wpisami innych procesów,
- wstawienie/odczyt danych o rozmiarze niewiększym od bloku 4096B jest operacją niepodzielną,
- dostęp do obiektów odbywa się na tej samej zasadzie, co do dostęp plików,
- można ustalić prawa dostępu do obiektu kolejki,
- dostępne są wywołania synchroniczne i asynchroniczne,
- w trybie synchronicznym i asynchronicznym zapis w przypadku braku odbiorców generuje SIG_PIPE,
- w przypadku braku nadawców funkcja odczytu zwraca natychmiast 0.

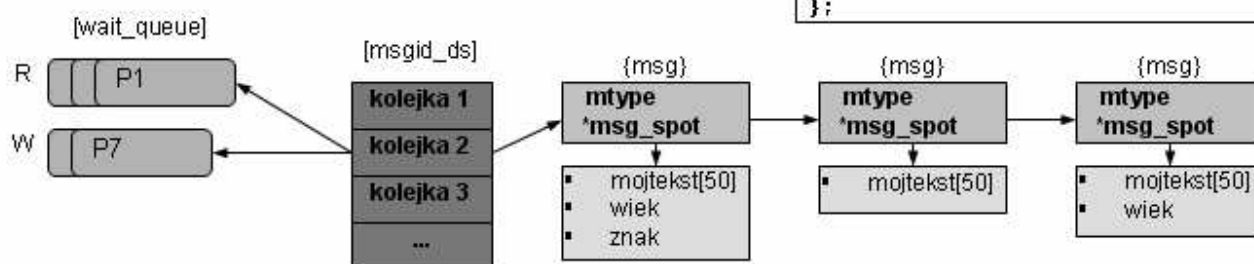
LINUX
- <code>mknod()</code> ;
- <code>mkfifo()</code> ;
- <code>open()</code> ;
- <code>close()</code> ;
- <code>read()</code> ; <code>fscanf()</code> ;
- <code>write()</code> ; <code>fprintf()</code> ;
- <code>unlink()</code> ;

4.12 Kolejki komunikatów

Kolejki komunikatów to najbardziej zaawansowany mechanizm w systemach unixowych. Charakteryzuje się elastycznością definiowania komunikatów, rozbudowanym mechanizmem odbioru komunikatów.

- identyfikacja kolejki: klucz (generowany w oparciu o plik i 8bitowy identyfikatora – `ftok()`),
- możliwe jest wystąpienie większej ilości wpisujących i odczytujących,
- istnieje możliwość adresacji (wybór typu wiadomości pozwala określić filtr dla odbiorcy),
- prawa dostępu – ustala się jak dla plików,
- maksymalna liczba kolejek w systemie: MSGMNI (128),
- maksymalny rozmiar komunikatu: MSGMAX (4096),
- maksymalny łączny rozmiar kolejki: MSGMNB (16384),
- zapisu do kolejki może dokonać każdy z dołączonych do kolejki procesów z odpowiednimi prawami,

- wprowadzany jest typ wiadomości – (long) mtype,
- odczyt pozwala na filtrację wiadomości wg wartości mtype:
 - = 0 – pierwszy komunikat w kolejce,
 - > 0 – pierwszy komunikat danego typu
 - < 0 – pierwszy komunikat najmniejszego typu, =< od -mtype,



```

/* struktura wysyłanej wiadomości */
struct msgbuf
{
    long mtype;           /* obowiązkowe */

    char mojtekst[50];    /* własne */
    int wiek;             /* dane */
    char znak;            /* programisty */
};

```

```

struct msgid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* first message enqueue, unsend */
    struct msg *msg_last; /* last message in queue, unsend */
    __kernel_time_t msg_stime; /* lastmsg send time */
    __kernel_time_t msg_rtime; /* lastmsg rcv time */
    __kernel_time_t msg_ctime; /* lastchange time */
    unsigned long msg_lcbytes; /* low 32 bits for 32 bit */
    unsigned long msg_lqbytes; /* data */
    unsigned short msg_cbytes; /* current number of bytes enqueue */
    unsigned short msg_qnum; /* number of messages in queue */
    unsigned short msg_qbytes; /* max number of bytes enqueue */
    __kernel_ipc_pid_t msg_lspid; /* pid of lastmsg send */
    __kernel_ipc_pid_t msg_lrpid; /* last receive pid */
};

```

LINUX <msg.h>
- msgget();
- msgctl();
- msgsnd();
- msgrcv();

4.13 Kolejki komunikatów POSIX

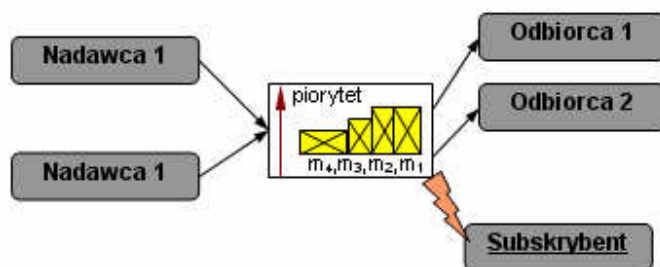
POSIX: Portable Operating System Interface

Mqueue to standaryzowana kolejka wiadomości (systemy unixsowe),

- jako jedyna IPC pozwala na definiowanie obsługi nadejścia wiadomości (puls sygnału),
- jako jedyna IPC pozwala na nadawanie priorytetów wiadomościom,
- wiadomości nie posiadają adresatów,
- możliwe jest wystąpienie większej ilości wpisujących i odczytujących,
- wiadomości o wyższym prioryecie wstawiane są bliżej początku kolejki,
- odczytanie wiadomości (1 proces) powoduje wyciągnięcie jej z kolejki,
- rejestracja sygnału odczytującego – tylko jeden dla danej kolejki,
- kolejki mocowane są w systemie plików – dostęp może być określony prawami,
- zapis do zapełnionej kolejki jest blokowany do momentu odebrania danych,
- odczyt jest blokowany, jeśli kolejka jest pusta.

Rejestracja powiadomienia wymaga podania funkcją mq_notify() mechanizmom danej kolejki sygnału systemowego (np. SIGUSR1) oraz wcześniejszego zarejestrowania funkcji obsługującej ten sygnał.

LINUX, np. QNX <mqueue.h>
- mq_open();
- mq_getattr();
- mq_setattr();
- mq_send();
- mq_receive();
- mq_notify();
- mq_close();
- mq_unlink();



4.14 Sygnały (UNIX, LINUX)

Sygnały - podstawowy system komunikacji między procesem a systemem oraz między procesami w systemach unixowych. Sygnały stanowią przerwania programowe, wywoływane w przypadku:

- wystąpienia błędów w trakcie wykonywania programu (nieprawidłowe odwołania, współpracy z systemem),
- zdarzeń zewnętrznych – asynchronicznie przekazywanych przez jądro komunikatów od innych procesów.

Każdy proces ma ustalone domyślne zachowanie w przypadku otrzymania sygnału. Zachowanie to może być zmienione programowo.

Struktura sygnału

Sygnały oprócz akcji przekazują pewną strukturę danych (`siginfo_t`). W tej strukturze zawierają się min.:

- numer sygnału (`si_signo`) - klasyfikujący powód jego wysłania, jest to liczba całkowita z zakresu 1-31; w rozszerzeniach systemów czasu rzeczywistego numery sygnałów mają rozszerzony zakres – do 63
- kod rozszerzony błędu (`si_code`),
- kod błędu (`si_errno`):
 - o większy od zera oznacza powód wysłania sygnału,
 - o równy zero (lub mniejszy) – sygnał wysłał użytkownik o UID `si_uid` z procesu `si_pid`.

```
typedef struct
{
    int si_signo;
    int si_code;
    union sigval si_value;
    int si_errno;
    pid_t si_pid;
    uid_t si_uid;
    void *si_addr;
    int si_status;
    int si_band;
} siginfo_t;
```

Wysyłanie i odbieranie sygnałów

Podczas wykonywania procesu, przed każdym przejściem z pracy w trybie jądra do pracy w trybie użytkownika, sprawdzane jest, czy nie nadeszły jakieś sygnały. Jeśli tak, następuje wykonanie akcji związanej z danym sygnałem. Zaistniałe zdarzenie jest zaznaczane poprzez zapalenie bitu odpowiedniego mu numeru. Wynika z tego fakt, że sygnały nie są kolejkowane, nie są zliczane i proces nie będzie znać liczby otrzymanych komunikatów, jeśli nadejdzie ich większa ilość, zanim zdąży zostać wykonana procedura obsługi. W strukturach danych wpisane będą parametry ostatniego wywołania.

Obsługa sygnałów

Otrzymany sygnał może być obsługiwany w jeden z następujących sposobów:

- IGN – zignorowanie sygnału,
- TERM – zakończenie procesu,
- STOP – zatrzymanie procesu,
- CORE – zakończenie procesu i złożenie obrazu pamięci na dysku,
- CUST – wykonanie procedury obsługi zdefiniowanej przez użytkownika.

Sprzęg systemowy obsługi sygnałów

Do zarządzania sygnałami wykorzystuje się:

```
typedef void (*sighandler_t)(int); // deklaracja wskaźnika funkcji obsługi sygnału
sighandler_t signal(int signum, sighandler_t handler); // funkcja zmieniająca procedurę obsługi sygnału
```

Wysyłanie sygnału do bieżącego i obcego procesu:

```
int raise(int sig); // wysłanie sygnału nr sig do bieżącego procesu
int kill(pid_t pid, int sig); // wysłanie sygnału nr sig do procesu o danym pid
```

Inne funkcje:

```
sigsuspend(), sigpending(), sigpause(), sigprocmask(), sigaction(), sigsetops(), sigvec(), sigsend(),...
```

Przegląd wybranych sygnałów

Nazwa	Nr	Opis	Akcja
SIGHUP	1	Zerwanie połączenia z terminalem	TERM
SIGILL	4	Wykonanie niedozwolonej instrukcji	CORE
SIGKILL	9	Zakończenie bezwarunkowe – nie da się zmienić procedury obsługi	TERM
SIGPIPE	13	Próba zapisu do potoku bez odbiorcy	TERM
SIGALRM	14	Odmierzenie czasu zadanego funkcją alarm()	TERM
SIGUSR1	16	Sygnał 1, zdefiniowany przez użytkownika	TERM
SIGUSR2	17	Sygnał 2, zdefiniowany przez użytkownika	TERM
SIGURG	21	Zajście zdarzenia pilnego	IGN
SIGSTOP	23	Zatrzymanie procesu	STOP

SIGRTMIN : SIGRTMAX - zakres sygnałów dla rozszerzeń czasu rzeczywistego.

4.15 Komunikaty systemowe Windows

Konstrukcja aplikacji w systemie Windows oparta jest o przetwarzanie komunikatów. Przewiduje ona konstrukcję głównego wątku procesu jako pętli obsługującej komunikaty wysłane przez system lub przez inne aplikacje. Możliwe jest oczywiście tworzenie procesów o charakterze konsoli lub wsadowym, ale nie są wówczas dostępne liczne mechanizmy systemowe (warto jest wówczas stworzyć aplikację okienkową o pojedynczym, niewidocznym oknie). Z każdym oknem związana jest procedura obsługi komunikatów. W procedurze tej jest realizowana obsługa poszczególnych komunikatów, związanych z tym oknem jako procesy interakcyjne lub systemowe (np. obsługa połączeń gniazdkowych).

```

LRESULT CALLBACK WindowProcedure (HWND hwnd,UINT message,
                                   WPARAM wParam, LPARAM lParam )
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage (0);
            break;
        default:
            return DefWindowProc (hwnd,message,wParam,lParam);
    }

    return 0;
}

```

Wybrane komunikaty:

- WM_PAINT
- WM_CLOSE
- WM_TIMER
- WM_CHAR
- WM_MOUSEMOVE
- WM_LBUTTONDOWN
- WM_KEYDOWN
- WM_QUIT
- WM_SIZE
- WM_USER
- WM_COMMAND
- ...

```

int WINAPI WinMain (HINSTANCE hinst, HINSTANCE hp, LPSTR arg,int n)
{
    HWND hwnd;
    MSG msg;

    WNDCLASSEX wincl;

    ..
    wincl.lpszClassName = "Aplikacja1";
    wincl.lpfnWndProc = WindowProcedure;
    ..
    if (!RegisterClassEx (&wincl))
        return 0;

    hwnd = CreateWindowEx (0,"Aplikacja1", "Tytuł", ...

    ShowWindow (hwnd, SW_SHOW);

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }

    return messages.wParam;
}

```

Komunikacja międzyprocesowa z użyciem przekazywania komunikatów okien

Wysyłanie wiadomości o rozmiarze `rozmiar`, przechowywanej pod adresem `adres`.

```
COPYDATASTRUCT cds;
HRESULT hResult;

cds.dwData = TYP_WIADOMOSCI;
cds.cbData = rozmiar;
cds.lpData = adres;

hWndDispatch = FindWindow( "MojaApl", "OknoDocelowe" );
if( hWndDispatch != NULL )
    SendMessage( hWndDispatch, WM_COPYDATA, (LPARAM) (HWND) hWnd, (LPARAM) (LPVOID) &cds);
```

Odbieranie wiadomości

```
COPYDATASTRUCT *pcds;

LRESULT CALLBACK WindowProcedure(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam )
{
    switch (message)
    {
        case WM_COPYDATA:
            pcds = (*COPYDATASTRUCT) lParam;
            if( pcds->dwData == TYP_WIADOMOSCI )
                PrzetwarzajDaneWiadomosci(cds);
            break;

        case ...
    }
    return 0;
}
```

Struktura opisująca wiadomość oraz obszar danych są kopiowane do procesu odbiorcy – kopia istnieje tylko do momentu zakończenia obsługi komunikatu `WM_COPYDATA`.

4.16 Schowek (Clipboard), Windows

Schowek to zarządzany przez system obszar pamięci do przechowywania danych o zarejestrowanym wcześniej typie. W schowku naraz może być jeden obiekt z danymi. Aplikacje mogą nasłuchiwać, kiedy pojawi się wybrany / umówiony typ danych.

Schowek obsługuje zdefiniowane typy danych. Przykładowe standardowe typy danych:

- `CF_BITMAP`
- `CF_TEXT`
- `CF_WAVE`
- `CF_HDROP`

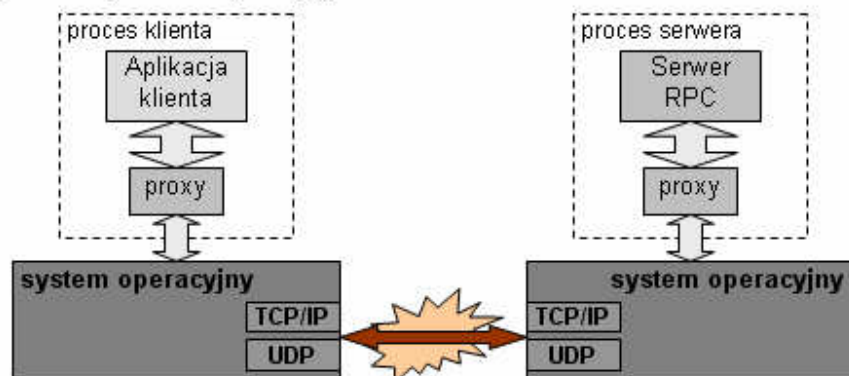
Możliwe jest definiowanie własnych typów danych.

Do obsługi schowka przeznaczono min. następujące funkcje:

<code>OpenClipboard()</code>
<code>OpenClipboard()</code>
<code>CloseClipboard()</code>
<code>GetOpenClipboardWindow()</code>
<code>GetClipboardData()</code>
<code>SetClipboardData()</code>
<code>RegisterClipboardFormat()</code>
<code>IsClipboardFormatAvailable()</code>

4.17 Zdalne wywołanie procedury (RPC)

Zdalne wywołanie procedury jest techniką umożliwiającą wywołanie przygotowanego podprogramu na lokalnej (LRPC) lub zdalnej (RPC) maszynie. Działa ona w oparciu o model klient-serwer. Technika ta jest zaimplementowana i wykorzystywana w większości systemów operacyjnych.



Rys. 4.9 Model RPC

Dane:

- dane są transformowane do formatu XDR (eXternal Data Representation), co ma zapobiec błędnej interpretacji przez różne procesory (Intel, Motorola) i systemy operacyjne,
- przekazywane dane muszą być interpretowane poprawnie – dotyczy to przede wszystkim wskaźników – proces poprawnego przekazywania wskaźników nazywa się przetaczaniem (ang. marshalling).

Funkcje:

- są specyfikowane przez pewien interfejs (nazwy funkcji, numer funkcji (adres), parametry wywołania),
- w każdej instancji systemu operacyjnego serwer RPC specyfikuje udostępniane funkcje przez następujące numery:
 - programu – identyfikator zestawu procedur,
 - procedury – kolejność procedury w zestawie,
 - wersji – reprezentujące wersję zestawu procedur (parametry).

Do implementacji serwera i klienta wymagane jest stworzenie specyfikacji zestawu funkcji (interfejs) w postaci nagłówka w języku C albo języku IDL, który to, przy pomocy odpowiednich narzędzi, jest tłumaczony na nagłówki dostępnych języków. W podobny sposób można stworzyć biblioteki konieczne do prawidłowego przetaczania danych (proxy). Realizacja komunikacji sieciowej RPC przebiega przy użyciu protokołu UDP lub TCP.

Metody mogą być wykonane w trzech trybach:

- niepewnym – bez gwarancji wykonania (UDP),
- co najmniej raz – bez ochrony przed niezamierzonym, wielokrotnym wywołaniem (UDP, implementacja potwierdzenia),
- dokładnie raz – najbardziej pożądana wersja (TCP).

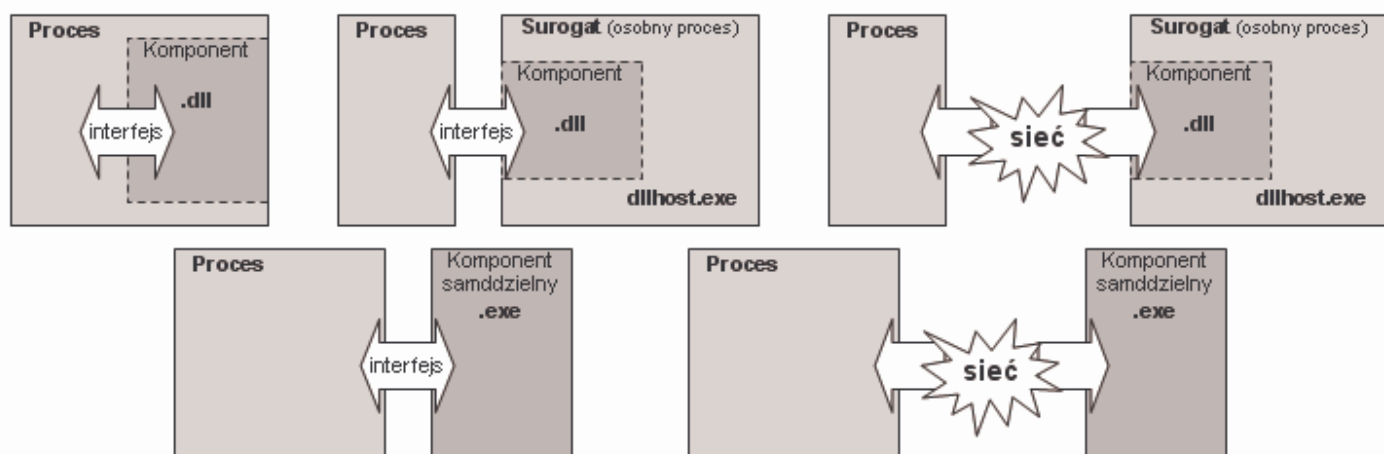
Wywołania metod mogą być:

- synchroniczne,
- asynchroniczne (wywołanie nie może zwracać wyniku),
- z odwołaniem zwrotnym,
- rozgłoszeniowe – do wszystkich serwerów w sieci, implementujących zestaw metod.

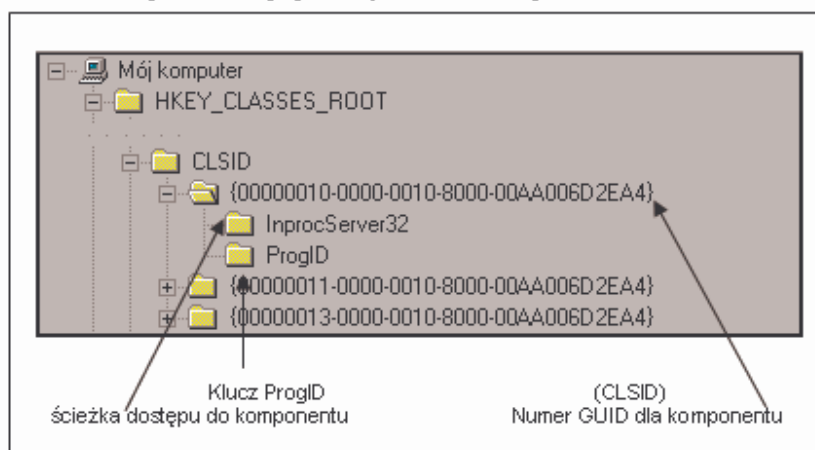
UNIX	LINUX	Windows
- callRPC()	- svc_register()	- RpcServerRegisterIf()
- registerRPC()	- svc_unregister()	- RpcServerUnregisterIf()
- svc_run()	- svc_sendreply()	- RpcServerListen()
- clnt_call()	- svc_run()	- RpcServerSubscribeForNotification()
- clnt_create()	- clnt_call()	- ...
- clnt_control()	- clnt_create()	
- clnt_destroy()	- clnt_control()	
- clnt_broadcast()	- clnt_destroy()	

4.18 COM/DCOM

COM to zintegrowana z systemem operacyjnym warstwa oprogramowania komponentowego. Oparta jest na obiektowej wersji RPC. Komponenty mogą komunikować się poprzez wyspecyfikowany interfejs. Owa komunikacja pozwala na wymianę danych między procesem a biblioteką dynamicznie ładowaną lub między dwoma procesami. Rozszerzenie DCOM realizuje komunikację w sieci (TCP, http tunneling, ...). Dzięki specyfikacjom interfejsów w języku IDL możliwe jest tworzenie komponentów w różnych językach programowania i w różnych pakietach (środowiskach). W środowisku Linux dostępne są łączniki DCOMa – EntireX.



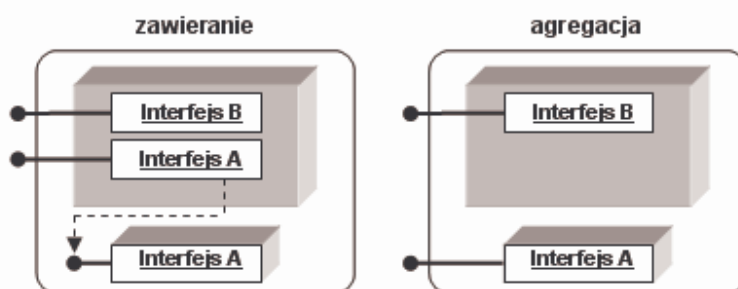
Rys. 4.10. Tryby funkcjonowania komponentów COM



Rys. 4.11. Dane komponentów, przechowywane w rejestrze systemowym.

```
interface IUnknown
{
    HRESULT QueryInterface( [in] REFIID riid, [out, iid_is(riid)] void **ppvObject);
    ULONG AddRef();
    ULONG Release();
};
```

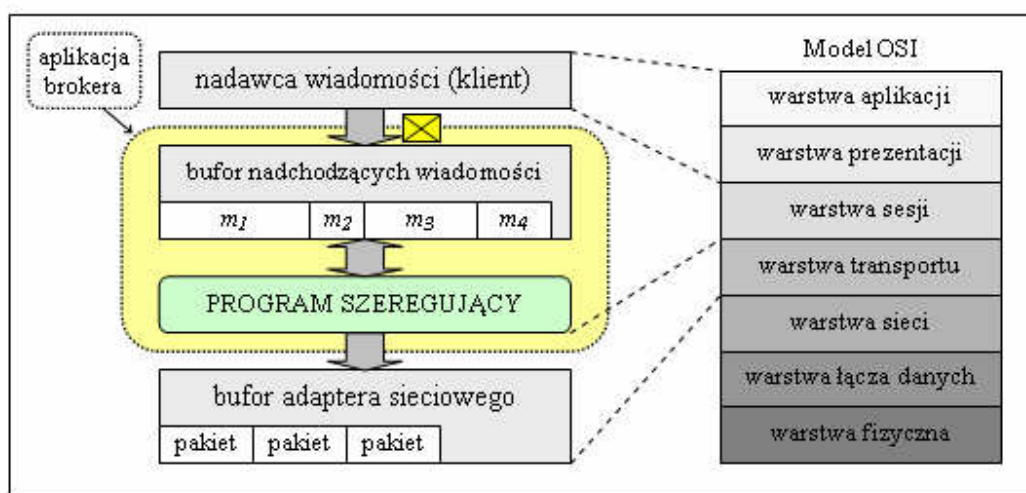
```
interface IPrzyklad : IUnknown
{
    import "unknwn.idl";
    HRESULT Oblicz([ in ] float wektor[256],
                  [ out ] float *wynik);
};
```



Rys. 4.12. Zawieranie a agregacja komponentów

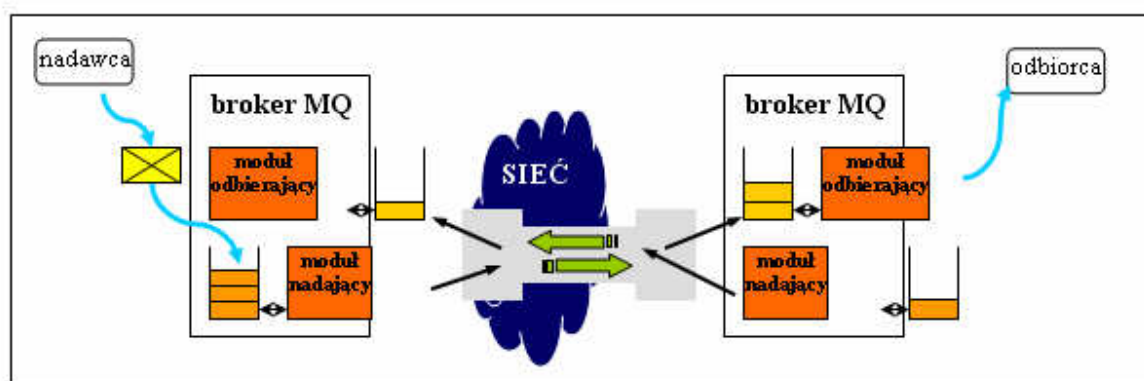
4.19 Systemy wiadomości kolejkowych

Implementacja komunikacji asynchronicznej nie jest trywialna. Proponuje się zatem gotowe rozwiązania dla różnych platform systemowych w postaci tzw. oprogramowania warstwy pośredniej (ang. middleware), zwanego systemem wiadomości kolejkowych (ang. MQ – Message Queuing). Oprogramowanie to pośredniczy w asynchronicznym przesyłaniu wiadomości między aplikacjami. Nadawca (aplikacja użytkownika) powierza swoje dane do systemu za pośrednictwem określonego interfejsu. Proces (serwer) odpowiedzialny za odpowiednie przekazywanie wiadomości nazywa się **brokerem wiadomości**. Aplikacja użytkownika powierza systemowi wiadomości kolejkowych dane, podając adres odbiorcy. Od momentu potwierdzenia nadania dostarczenie wiadomości jest obowiązkiem systemu kolejkowego. Obsługuje on także wszelkie zagadnienia transportowe z tym związane. Dla nadawcy systemy wiadomości kolejkowych oferują obsługę pięciu najniższych warstw modelu OSI (rys. 4.13). Znacznie ułatwia to tworzenie systemów rozproszonych, skracając czas projektowania, implementacji i testowania.

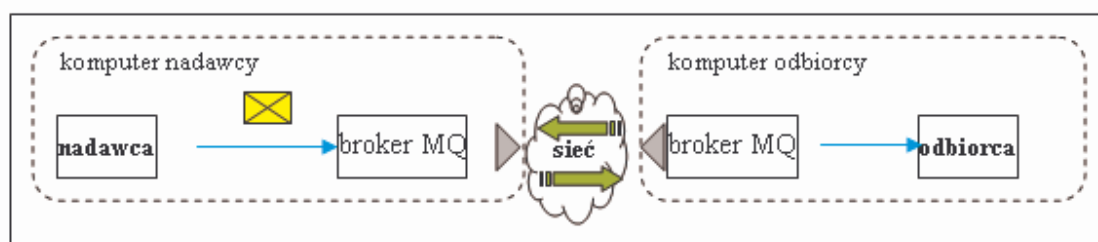


Rys. 4.13. System wiadomości kolejkowych w kontekście modelu OSI

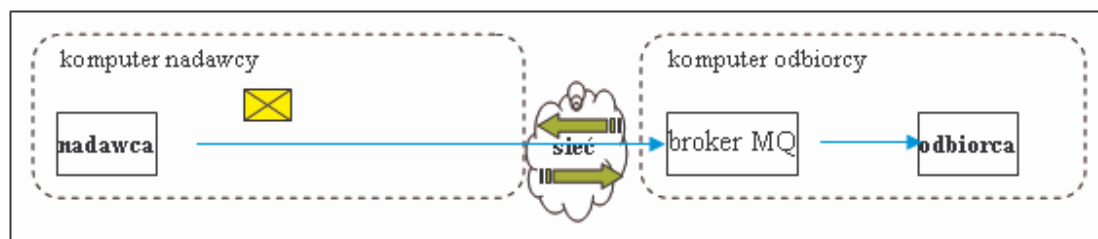
Wiadomość stanowi dla nadawcy pewną interpretowalną spójną paczkę danych (przetwarzaną w warstwie prezentacji i aplikacji). Zostaje ona przekazana do brokera, który nawiązuje połączenie z brokerem docelowym, łącząc się z nim po wybranym medium, ustalonym protokołem dokonując przekazania danych (warstwy sesji, transportu i sieci). Większość brokerów oferuje szeregowanie wiadomości zalegających w kolejce do wysłania. Celem danego szeregowania może być uwzględnianie pilności nadania (priorytet) czy też optymalizacja parametrów pracy systemu.



Rys. 4.14. System wiadomości kolejkowych w sieci



Rys. 4.15. Model klienta niezależnego



Rys. 4.16. Model klienta zależnego

Implementacje:

MQSeries (IBM), MSMQ (Microsoft), standard JMS (Sun), implementacje JMS – np iPlanet.

Standard JMS jest zbiorem interfejsów w języku Java do komponentów oferujących usługi systemu wiadomości kolejkowanych. Jest on jednakowy dla wszystkich implementacji. Przykładowo, definicja metody wstawiającej wiadomość do kolejki jest następująca:

```
void send(Message message)
```

Poniżej przedstawiony jest przykładowy kod, wykonujący wstawienie wiadomości do kolejki nadawczej.

```
String stockData; /* Stock information as a string */  
TextMessage message;  
/* Set the message's text to be the stockData string */  
message = session.createTextMessage();  
message.setText(stockData);  
  
/* Send the message */  
sender.send(message);
```

Standard JMS przestrzega kolejności powierzonych wiadomości [JMS]. Wiadomość o wyższym priorytecie zostaje wysłana wcześniej, jednakże wiadomości o tym samym priorytecie są wysyłane według kolejności nadejścia (FIFO RT).

Pamięć operacyjna - pamięć, w której znajdują się aktualnie wykonywane przez system procesy oraz ich dane.

Pamięć masowa - pamięć, w której nie można umieścić kodu wykonywanego w systemie, a w której przechowywane są dane procesów lub obrazy procesów.

Pamięć wirtualna – technika pozwalająca na obsługę (sprzętową/programową) wirtualnej przestrzeni adresowej, przekraczającej rozmiar dostępnej pamięci fizycznej.

5.1 Adresacja pamięci fizycznej

Komputery 8-bitowe: pamięć liniowa, adres = 2 bajty – młodszy i starszy – adresowany obszar: 2^{16} B = 64KB (0-65535)

Adres absolutny (fizyczny) - jednoznaczny identyfikator komórki w pamięci fizycznej.

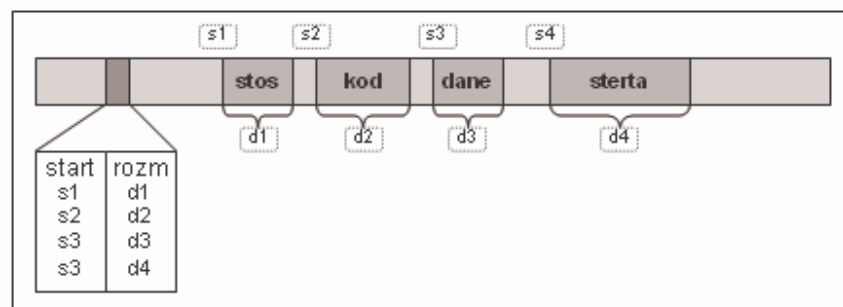
Adres wirtualny – adres komórki pamięci w wirtualnej przestrzeni adresowej systemu.

Adres logiczny – relatywny adres komórki pamięci (dla procesora), określane najczęściej jako SEGMENT:OFFSET

5.2 Segmentacja pamięci

Podział obszaru procesu na segmenty - jednostki semantyczne: np. blok kodu, blok danych, stos, sarta,...

Dane segmentu: początek i długość.



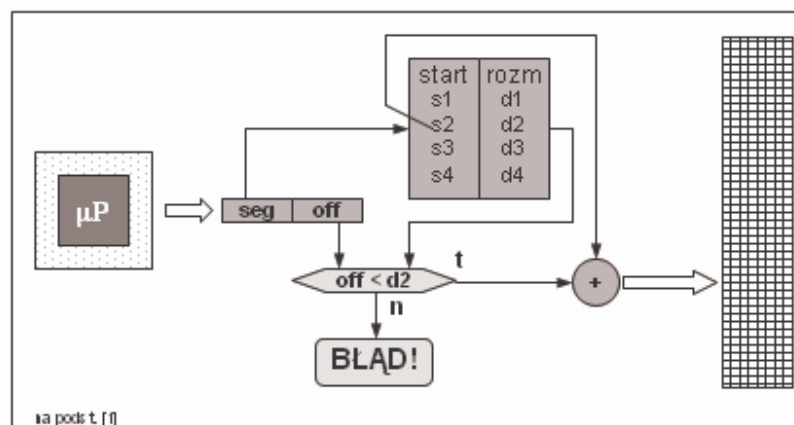
Tryb dostępu do segmentu: odczyt / wykonywanie.

Realizacja wyznaczania adresu w oparciu o:

- rejestry bazowe (szybkie, ograniczone) – np.: Intel: CS – Code Seg., DS. – Data Seg., SS – Stack Seg., ES – Extra Seg.;
- tablicę rejestrów bazowych (odczyt komórki: narzut odwołań do pamięci 200%, zmniejszany do ok. 115% poprzez użycie rejestrów asocjacyjnych (ostatnio używane wartości)).

Tablice deskryptorów segmentów (Intel):

- globalna - GDT (Global Descriptor Table) - systemowa, zawiera adresy tablic lokalnych,
- lokalna - LDT (Local Descriptor Table) - lokalna tablica deskryptorów, zawierająca informacje o pozostałych segmentach.



5.3 Przydział ciągły

Strategie przydziału wolnego obszaru:

- pierwsze dopasowanie,
- najlepsze dopasowanie,
- najgorsze dopasowanie.



Fragmentacja zewnętrzna – przydziały pamięci pozostawiają liczne ale małe bloki pamięci wolnej, straty nawet 50%

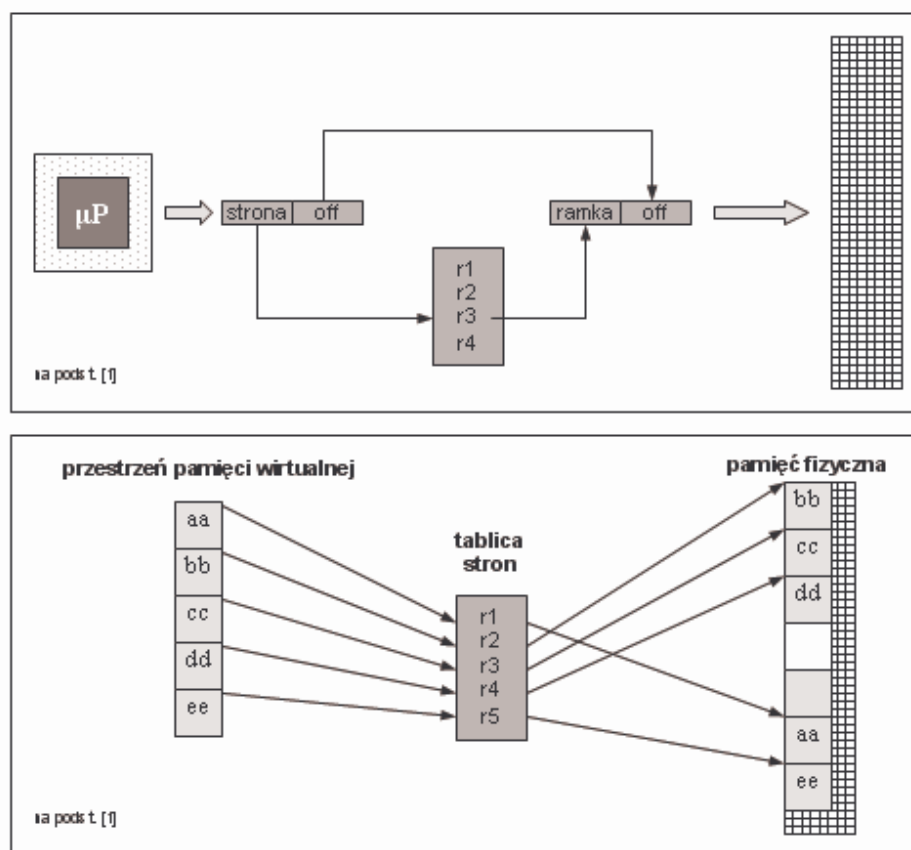
Fragmentacja wewnętrzna – przydzielony blok jest większy od zamawianego (różnica niewielka, koszt przechowywania informacji o dziurze byłby większy).

Rozwiązanie problemu fragmentacji zewnętrznej – upakowanie (relokacja bloków, jeśli to możliwe).

5.4 Pamięć stronicowana

Stronicowanie pamięci – podział pamięci na strony o stałym rozmiarze (2^n B). Stronie (adres logiczny) odpowiada ramka (adres fizyczny). Adresacja: <nr strony, przesunięcie>.

Wspomaganie sprzętowe – nr strony jest odwzorowywany na nr ramki, przesunięcie bez zmian.



Nie występuje fragmentacja zewnętrzna, natomiast powszechna jest fragmentacja wewnętrzna.

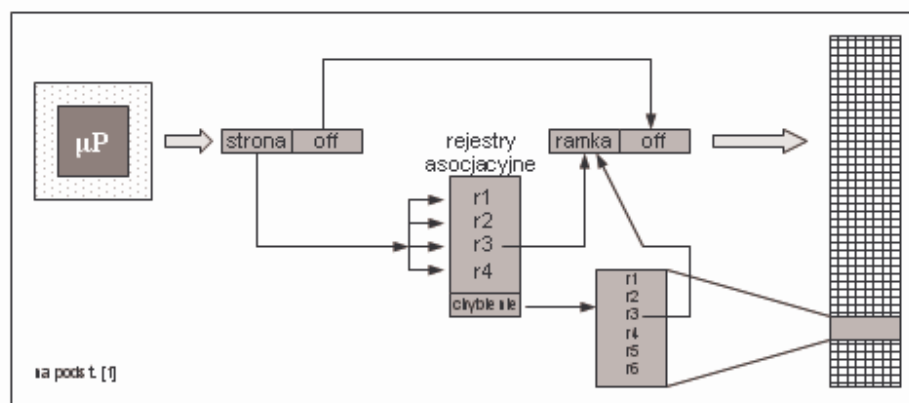
Ochrona dostępu do pamięci: strona może mieć tryb tylko do odczytu / do zapisu i odczytu / do wykonywania. Dodatkowe ograniczenie dostępu poprzez atrybut bitu poprawności (fragmentacja wewnętrzna!).

Strony dzielone (pamięć dzielona, biblioteki systemowe i inne (kod wznowialny))

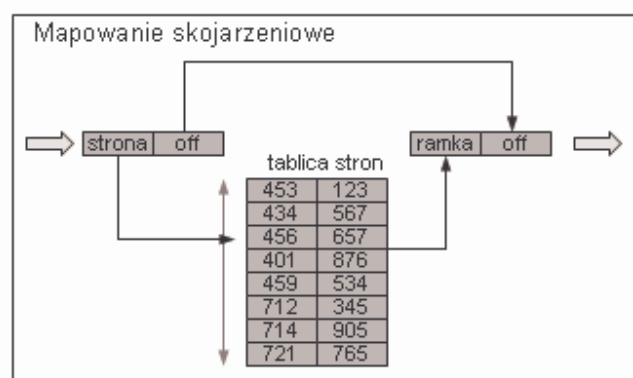
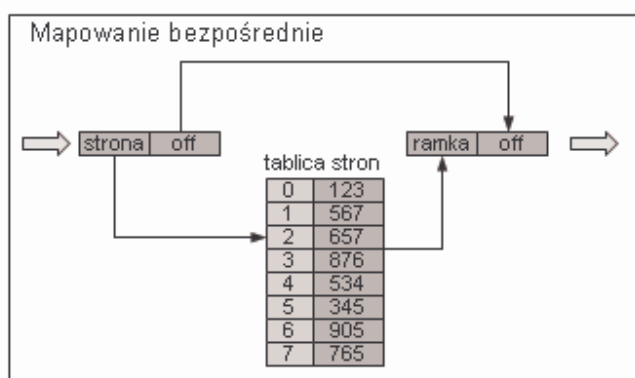
Realizacja adresacji w oparciu o tablice stron

Strony dostępne dla procesu są umieszczone w tablicy. W bloku kontrolnym procesu umieszcza się do niej wskaźnik, gdyż jej rozmiar może być bardzo duży. Sprzęt może dostarczać wsparcie w postaci rejestru adresującego taką tablicę.

Ze względu na dwukrotne odwołania do pamięci w przypadku pojedynczego odczytu, używa się rejestrów asocjacyjnych (szybkich, aczkolwiek ograniczonych do niewielkiej liczby wpisów), które przechowują najczęściej wywoływane strony i odpowiadające im ramki.

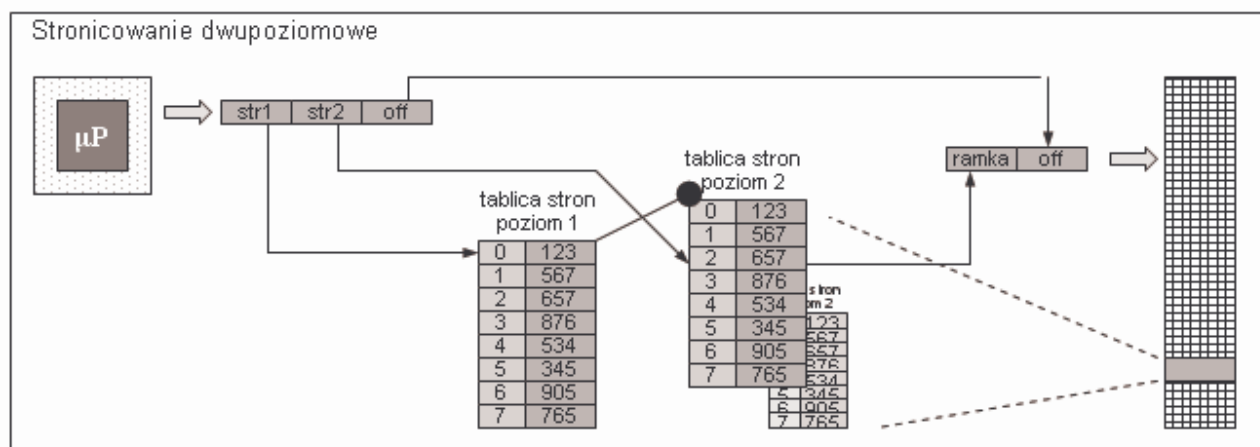


Mapowanie bezpośrednie i skojarzeniowe



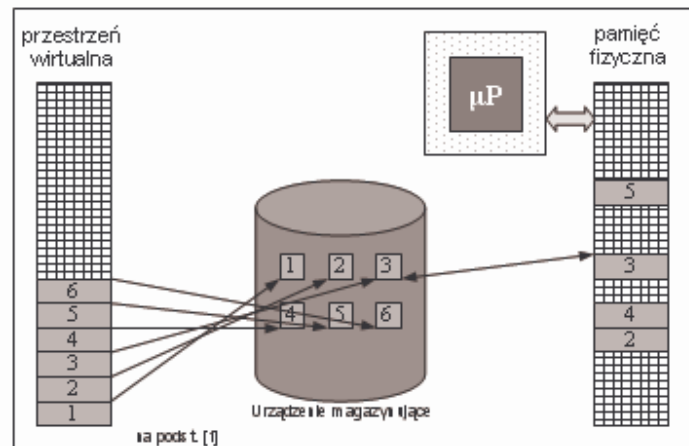
Stronicowanie wielopoziomowe

Spotykane jest stronicowanie dwu- i trójpociomowe



5.5 Obsługa pamięci wirtualnej

Wymiana stron pamięć <-> dysk



Stronicowanie na żądanie – ładowanie stron, które będą potrzebne do wykonania programu

Procedura leniwej wymiany – wymiana stron ma miejsce w momencie, kiedy są one potrzebne do wykonywania kodu.

Brak strony (page fault) – brak strony w pamięci w trakcie wykonywania programu – pociąga za sobą procedurę ładowania.

Bit poprawności – 1 - strona dostępna, 0 – strona niedostępna (na dysku lub niezaalokowana do obszaru adresowego).

Bit zmiany zawartości (aktualności) – 1 - strona zgodna z obrazem w pliku wymiany, 0 – strona zmodyfikowana.

Bit odniesienia – początkowo 0, ustawiany na 1 po użyciu choć jednego bajta z tej strony (odniesienia się do tej strony).

Zastępowanie stron

Algorytmy zastępowania stron:

- FIFO (-> anomalia Belady'ego)
- OPT (zastępowanie strony, która będzie najdłużej nieużywana – wizjonerstwo;)
- LRU (Last Recently Used – zastępowanie strony, która była najdłużej nieużywana)
 - o implementacja w oparciu o liczniki czasu dla stron (sprzętowe)
 - o implementacja w oparciu o stos
- przybliżające LRU
 - o algorytm dodatkowych bitów odwołań – bajt z periodycznie ustawianymi bitami użycia strony
 - o algorytm drugiej szansy – wybór strony do zrzutu na dysk w oparciu o FIFO z bitem odniesienia
 - o algorytm drugiej szansy + bit zmiany zawartości.

Algorytm drugiej szansy (Strategia zegarowa) (bufor okrężny, ~zegar)

- każda strona ma bit odniesienia,
- w buforze okrężnym są zgromadzone dane stron,
- każde odwołanie do strony/ramki ustawia bit wykorzystania na 1,
- w trakcie wymiany strony przeszukiwane są kolejne strony:
 - wymieniana jest pierwsza znaleziona z bitem wykorzystania =0;
 - wszystkie przeszukane po drodze mają zmniejszoną wartość z 1 na 0.

Obszar roboczy – Working Set – zestaw stron, które są używane przez program (przez pewien okres do teraz)

Szamotanie procesu – więcej czasu trwa obsługa stronicowania dla procesu niż jego wykonywanie.

5.6 Obsługa pamięci w systemie Linux

Przestrzeń adresowa procesu podzielona na 2 części:

0 – 3GB – przestrzeń procesu

3 GB – 4 GB – przestrzeń dostępna w trybie jądra, współdzielona.

PAE - Physical Address Extension – 64 GB – rozszerzenie dla procesorów Intel Pentium Pro +,

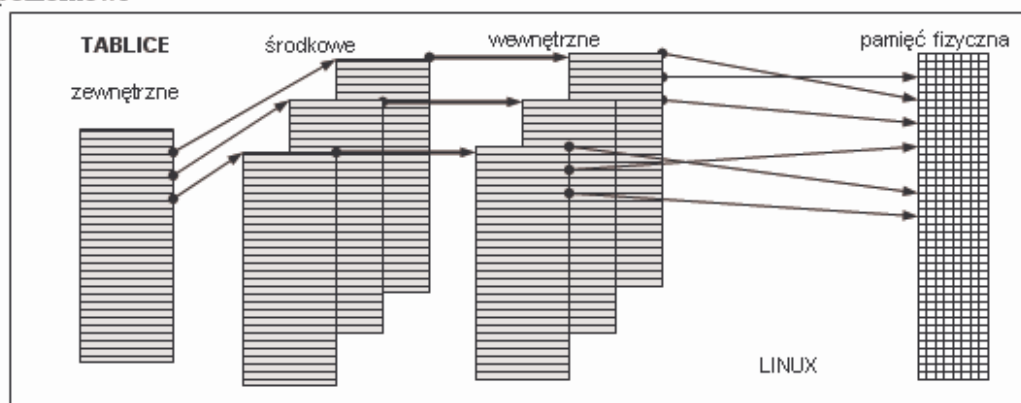
Strefy: zwykła, HIGHMEM, DMA.

Ramka 4KB (Intel), katalog stron zajmuje jedną stronę 2^{10} adresów po 4B.

Parametry strony:

- blokada ramki
- powodzenie operacji I/O
- bit odniesienia (alg LRU, zerowany po przeszukaniu stron do przeniesienia na dysk)
- bit zmiany zawartości

Stronicowanie trójpoziomowe



Tablice środkowe są w bieżącej implementacji ograniczone do jednego wpisu, procesory Intel Pentium – wsparcie dla adresacji dwupoziomowej, procesor DEC Alpha - architektura 64bit, przestrzeń > 4GB, wsparcie adresacji trójpoziomowej.

Postarzanie stron – w Linuxie stosuje się algorytm przybliżony do LRU, postarzanie odbywa się okresowo, w oparciu o przesunięcie (dzielenie przez 2, 8bitów), strona o wartości 0 jest uważana za starą, przenoszona jest z listy stron aktywnych do listy stron nieaktywnych, dzielonych dodatkowo na 'brudne' i 'czyste'.

Strona zawierająca kod (tryb wykonywalny) – powiązana jest z i-nodem odpowiedniego pliku, zapisanego w cache i-nodów

Strony anonimowe (np. sterła) – przechowywane są w pliku wymiany (partycja swap).

Wyszukiwanie stron w tabelach odbywa się przy użyciu posortowanej listy lub drzewa czerwono-czarnego.

Zarządzanie wolnymi obszarami pamięci fizycznej

Linux posiada mechanizm obsługi mapowania ciągłych bloków stron na ciągły obszar pamięci fizycznej (kolejne ramki). Działa on dla pamięci głównej, ale także dla pamięci jądra (jedna przestrzeń adresowa!). W systemie informacje o wolnych obszarach pamięci są przechowywane w tablicy indeksowanej kolejnymi potęgami 2.

Algorytm kumpla

- W przypadku zamówienia bloku pamięci: przydzielany jest najmniejszy pasujący blok pamięci, jeśli takowy jest wielokrotnie większy (2^n) to następuje rekurencyjne dzielenie bloku na połowy, wydzielone wolne obszary trafiają do niższych indeksów w tablicy.
- W przypadku zwolnienia bloku pamięci: wyszukiwany jest rekurencyjnie 'kumpel', tzn. sąsiadujący blok o tym samym rozmiarze i obydwa łączy w jeden o 2x większym rozmiarze.

5.7 Obsługa pamięci w systemie Windows NT

Win NT udostępnia procesom liniowy obszar adresowy pamięci wirtualnej o rozmiarze 4GB.

W pierwotnej koncepcji pierwsze 2GB pamięci przydzielane jest użytkownikowi, wyższe 2GB – tryb jądra, możliwe jest przesunięcie na 3GB-1GB.

NULL 64KB	stronicowane 000000-7fffffff	64KB !_ptr	bezpośrednio mapowane 80000000-bfffffff	stronicowane c0000000-	niestronicowane -fffffff
2GB - tryb użytkownika			2GB - tryb jądra		

Zarządzanie pamięcią wirtualną odbywa się przez wielowątkowy proces VMM (virtual memory manager).

Rozmiar strony/ramki jest ustalony na 4KB dla architektury Intel32, dla innych architektur może być 4-64KB (np. dla MIPS może być ustawiany – wybrano 8KB). Istotna jest obecność/brak sprzętowego wsparcia pamięci wirtualnej.

PAE - Physical Address Extension – 64 GB – rozszerzenie dla procesorów Intel Pentium Pro +,

AMD64 – 40/48 bitów przestrzeni adresowej,

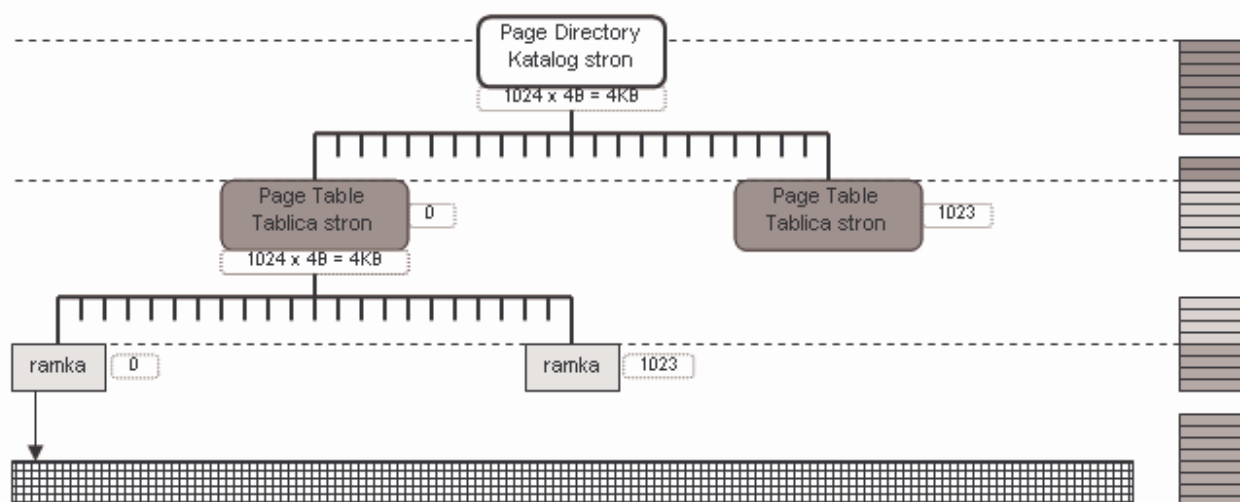
AWE - Address Windowing Extensions – rozszerzenie przestrzeni adresowej (przy zachowaniu 32-bitowych wskaźników!)

VAD – Virtual Address Descriptor – struktura służąca do zarządzania zaalokowanym obszarem pamięci.

TLB – Translation Lookaside Buffers – umieszczone w pamięci podręcznej procesora max 32 tłumaczenia adresów.

Algorytm wymiany wpisu w cache – zastępowany jest najmniej używany ostatnio wpis.

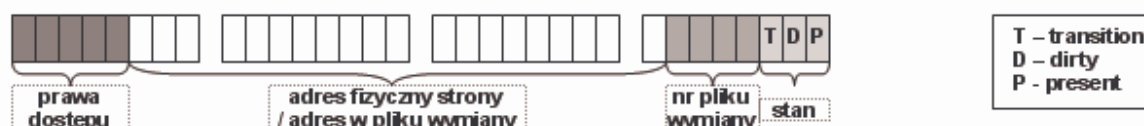
Realizacja adresacji wirtualnej



Katalog stron – jeden dla procesu.

Opis strony pamięci

Z każdą stroną w pamięci związane jest 4-bajtowe pole opisu stanu PTE (page table entry).



stan:

T	D	P	opis
0	-	0	Strona niewłaściwa
-	0	1	Strona prawidłowa
-	1	1	Strona prawidłowa, "brudna"
1	0	0	Strona w stanie przejściowym, zapisana
1	1	0	Strona w stanie przejściowym, "brudna"

prawa dostępu:
 - PAGE_NOACCESS
 - PAGE_READONLY
 - PAGE_READWRITE

Zarządzanie ramkami

Każdy proces ma określoną minimalną i maksymalną liczbę ramek, potrzebnych do wykonania. W razie zapełnienia całej pamięci fizycznej, system poszukuje wolnych ramek algorytmem przybliżonym do LRU. W przypadku skrajnym wszystkie procesy mają ograniczony zestaw ramek do wartości minimalnej. System tworzy dane statystyczne procesów w oparciu o liczbę błędów przy odwołaniu do stron.

W bazie danych ramek znajdują się informacje o stanie ramki:

- valid – ważna, używana przez aktywny proces, zgodna z obrazem w pliku wymiany,
- modified – zmodyfikowana, nie zapisana na dysk (TDP = 110),
- standby – zwolniona z zestawu procesu,
- free – wolna, czeka na wyzerowanie,
- zeroed – wolna, wyzerowana, czeka na przydział,
- bad – uszkodzona (błąd sprzętowy), wyłączona z użytku.

Baza danych składa się z sześciu list ramek, znajdujących się w wyżej wymienionych stanach. Pozwala to na szybkie szukanie ramek wolnych czy też poszukiwanie ramek do 'odzysku'.

Podstawowe pojęcia

Urządzenie znakowe – port -> port równoległy, port szeregowy, ...

Metoda PIO

Urządzenie blokowe – blok danych -> DMA, UDMA,

Magistrale: numeracja (identyfikacja), przydział zasobów, sterowanie, przełączanie

Buforowanie

Metoda odpytań (polling)

Przerwanie sprzętowe

- niemaskowalne

- maskowalne

Wywołania synchroniczne i asynchroniczne

Zegary czasu rzeczywistego, budziki

Dostęp do sprzętu w systemie MS-DOS

Dostęp do portów wejścia / wyjścia

W systemie MS-DOS nie ma ochrony dostępu do sprzętu (nie ma wielozadaniowości ;). Dlatego też operacje wejścia / wyjścia można wykonywać bezpośrednio bez narzutów systemowych, używając funkcji:

- IN / OUT w asemblerze lub wstawkach asemblerowych,
- inport() / outport() w języku C - <dos.h>

```
unsigned      inport  ( unsigned __portid );           // int = 2B!
unsigned char inportb ( unsigned __portid );
void          outport ( unsigned __portid, unsigned __value ); // int = 2B!
void          outportb( unsigned __portid, unsigned char __value );
```

Obsługa przerwania sprzętowych

W systemie MS-DOS można napisać własną procedurę obsługi przerwania w postaci pewnej funkcji. Do ustawienia jej używa się funkcji `getvect()` / `setvect()`. Dodatkowo można utrzymać tę procedurę po zakończeniu działania programu przy pomocy funkcji `keep()` (programy rezydentne).

```
void interrupt nowe_przerwanie(__CPPARGS);
void interrupt (*stare_przerwanie)(__CPPARGS);

void main(void)
{
    stare_przerwanie=getvect(0x1c);
    setvect(0x1c,nowe_przerwanie);
    keep(0, (_SS + (_SP/16) - _psp));
}

void interrupt nowe_przerwanie(__CPPARGS)
{
    asm
    {
        inc liczba
        mov ah,0x10
        // ..
    }
    outport(0x60,0xed);

    // stare_przerwanie ... mozna wywolac
}
```

Dostęp do sprzętu w systemie QNX

System QNX został zaprojektowany jako system czasu rzeczywistego dedykowany przede wszystkim na procesory z rodziny Intel i386. Ze względu na wykorzystywanie go w systemach sterowania został wyposażony w sprzęg programowy do obsługi sprzętu komputerowego, cechujący się prostą implementacją

Sprzęg systemowy

Specyfikacja interfejsu sterowników w systemie QNX jest zgodna z normami POSIX. Urządzenia są reprezentowane w systemie plików jako obiekty o ścieżce `/dev/urządzenie`.

Dostęp do sterownika od strony aplikacji klienckiej zgodnie z normą POSIX powinien być możliwy przy użyciu funkcji `open()`/`close()` (inicjacja i odłączenie), `read()`/`write()` (zapis i odczyt). Standard obejmuje też inne funkcje, np. `lseek()`. Przykładowo odczytanie i zapisanie wartości do portu urządzenia można dokonać następująco:

```
urz = open("/dev/urządzenie", O_RDWR);
read(urz, &dana, 1 /* bajt */);
write(urz, &dana, 1 /* bajt */);
close(urz);
```

Powiązanie sterownika ze ścieżką odbywa się podczas inicjacji sterownika. Jego proces można uruchomić wraz z systemem albo ręcznie, w dowolnym momencie.

Dostęp do portów wejścia / wyjścia

W systemie QNX dostęp do portów wejścia/wyjścia jest uproszczony ze względu na przeznaczenie systemu – instancje są najczęściej dedykowane pod konkretne rozwiązania i pozwala się na bezpośrednie operacje wejścia/wyjścia w wątkach, zakładając, że programista jest świadomy tego, co wykonuje program. System operacyjny nie implementuje wielodostępu. Aby użyć operacji pisania i czytania z portów we/wy (instrukcje IN, OUT) należy poinformować system operacyjny, że dany wątek będzie z nich korzystać przy pomocy funkcji `ThreadCtl()`. Funkcja ta wymaga uprawnień roota.

```
ThreadCtl(NT0_TCTL_IO, 0);
```

Obsługa przerwań sprzętowych w systemie QNX

Zgłoszenie przerwania sprzętowego w systemie QNX jest przejmowane bezpośrednio przez kernel. Przerywa on działanie aktualnie wykonującego się procesu / wątku i wywołuje procedurę obsługi przerwania ISR (*Interrupt Service Routine*). Procedura ta ma na celu dokonanie pewnych prostych operacji, np. odczyt danych ze sprzętu czy sprawdzenie pochodzenia przerwania. Jej priorytet jest wyższy od priorytetów jakichkolwiek wątków oprogramowania. W przypadku potrzeby przekazania informacji do programu wysyła sygnał `SIGEV_INTR`. Operacja ta zabiera nieznaczny czas procesora. Proces zainteresowany przerwaniem uruchamia wątek blokujący się oczekiwaniem na sygnał. Wątkowi temu można nadać dowolny priorytet, co umożliwia uprzywilejowaną w stosunku do pozostałych procesów obsługę przerwania czy też nieprzerwaną pracę ważniejszych procesów. Takie rozwiązanie jest zgodne z postulatami wymagań czasu rzeczywistego w stosunku do systemu operacyjnego. Istotna jest też możliwość obsługi jednego przerwania sprzętowego przez wiele procesów. Podobnie jednym procesem można obsługiwać kilka przerwania.

Od strony implementacji programista ma do wyboru dwa sposoby implementacji:

- użycie zdarzeń systemowych,
- stworzenie własnej procedury ISR.

Przypadek pierwszy jest prosty w implementacji. Wykorzystywana jest standardowa procedura ISR. Aplikacja zainteresowana obsługą przerwania powołuje wątek do jego obsługi. Nadaje on sobie odpowiednie prawa dostępu do sprzętu, podpinając pod konkretną linię przerwania wysłanie sygnału do siebie i przechodzi w stan oczekiwania. W stanie tym nie zajmuje zasobu procesora. W przypadku nadejścia przerwania wątek odbiera sygnał wykonując zadane operacje aż do ponownego przejścia w stan oczekiwania. Szkic kodu ilustrującego implementację problemu zamieszczony jest poniżej:

```
void * obsluga_przerwania (void * data)
{
    int id;
    ThreadCtl( _NTO TCTL_IO, NULL);           // nadanie odpowiednich praw wątkowi
    id = InterruptAttachEvent( nr_przerwania, &event, 0)
    if(id == -1)
        return;                               // przerwanie nie udało się przydzielić

    while (1)
    {
        InterruptWait (NULL, NULL);           // oczekiwanie na przerwanie

        // w tym miejscu należy wykonać operacje związane z obsługą przerwania

        InterruptUnmask(nr_przerwania, id);    // odblokowanie źródła przerwania
    }
}
```

Przypadek drugi dotyczy przypadków wymagających własnej procedury ISR. Procedura ta działa na poziomie kernela i jest przez niego uruchamiana w trybie natychmiastowym. Dotyczy to sytuacji, w których z przerwaniem związane są przychodzące dane, które należy odebrać jak najszybciej. W takiej sytuacji wątek obsługi nie musi mieć wysokiego priorytetu. Procedura ISR może być także wykorzystywana do filtrowania źródeł przerwania w przypadku większej ich liczby. Jej szkic implementacyjny podany jest poniżej:

```
void * obsluga_przerwania (void * data)
{
    int id;
    ThreadCtl( _NTO TCTL_IO, NULL);
    id = InterruptAttach( nr_przerwania, procedura_isr, NULL, 0, 0);
    if(id == -1)
        . . .
}
```

Procedura obsługi przerwania w tej metodzie będzie nieznacznie zmodyfikowana:

```
const struct sigevent* procedura_isr (void *arg, int id)
{
    // w tym miejscu można umieścić sprawdzenie źródła przerwania,
    // wykonać odwołania do sprzętu czy zablokować ponowną obsługę przerwania

    return (&event);
}
```

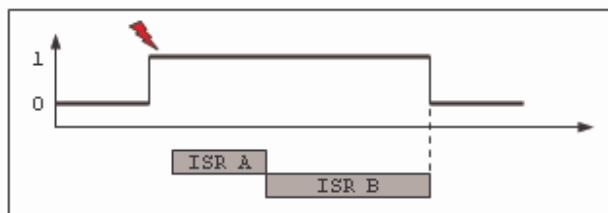
W zależności od uwarunkowania sprzętowego, przerwanie mogą mieć charakter wyzwalań:

- poziomem – wówczas sprzęt utrzymuje stan aktywacji przerwania aż do programowego skasowania tego stanu – procedura obsługi przerwania musi zawierać odblokowanie źródła przerwania;
- zboczem narastającym – wyzwolenie przerwania automatycznie kasuje jego źródło.

Przerwania współdzielone

Istnieje możliwość obsługi przerwania współdzielonych przez sprzęt (kilka urządzeń wykorzystuje to samo przerwanie). Wówczas pod daną linię podpinane są procedury obsługi przerwania. Każda z nich, związana z konkretnym sprzętem, sprawdza, czy to ona ma obsłużyć zdarzenie. W przypadku pojawienia się wysokiego poziomu przerwania uruchamiane są

kolejno owe procedury – po wykonaniu przez ostatnią odblokowywania źródła, poziom przerwania wraca do poziomu niskiego.



Dostęp do sprzętu w systemie Linux

Sprzęg systemowy

Sprzęg systemowy operacji wejścia/wyjścia w Linuxie jest analogiczny do sprzęgu w systemach unixowych, w tym QNX.

Dostęp do portów wejścia / wyjścia

W systemie Linux dostęp do portów wejścia/wyjścia jest uproszczony - pozwala się na bezpośrednie operacje wejścia/wyjścia w wątkach, zakładając, że programista jest świadomy tego, co wykonuje program. System operacyjny nie implementuje wielodostępu. Aby użyć operacji pisania i czytania z portów we/wy (instrukcje IN, OUT) należy poinformować system operacyjny, że dany wątek będzie z nich korzystać przy pomocy funkcji zawartych w <sys/io.h>:

- `ioperm()` – funkcja przydziela dostęp do określonego zakresu portów w obrębie przestrzeni 0x000-0x3FF
- `iopl()` – funkcja nadaje swobodny dostęp do wszelkich portów, możliwość wyłączania przerwań itp.

Funkcje te wymagają uprawnień roota.

```
int ioperm(unsigned long from, unsigned long num, int turn_on);
int iopl(int level);
```

// b – bajt, w – słowo (word), 2B, _p – opóźnienie 1mikrosekunda po operacji

```
inb (unsigned short int port);
inb_p (unsigned short int port);
inw (unsigned short int port);
inw_p (unsigned short int port);
outb (unsigned char value, unsigned short int port);
outb_p (unsigned char value, unsigned short int port);
outw (unsigned short int value, unsigned short int port);
outw_p (unsigned short int value, unsigned short int port);
```

Alternatywną metodą jest użycie sterownika do portów `/dev/port`, przy pomocy `lseek()` wybiera się port, zapis/odczyt funkcjami `write()` / `read()`.

Obsługa przerwań sprzętowych

System Linux umożliwia ustawianie własnych procedur obsługi przerwania. System obsługi przerwań jest trójpoziomowy:

- procedura niskopoziomowa,
- procedura użytkownika szybka (tzw. górna połowa),
- procedura użytkownika długa (tzw. dolna połowa).

Procedury niskopoziomowe są zdefiniowane w systemie. Każda linia przerwań ma przydzieloną jedną z trzech funkcji do obsługi:

- `bad_IRQNR_interrupt` - 'zaśleпка' dla przerwania nieobsługiwanego, ogranicza się do odblokowania źródła,
- `IRQNR_interrupt` – obsługa z blokowaniem innych przerwań i z zachowywaniem wszystkich rejestrów,
- `fast_IRQNR_interrupt` - obsługa bez blokowania pozostałych przerwań, zachowanie wybranych rejestrów.

Dopisanie nowej, własnej obsługi przerwania do listy procedur obsługi przerwania (jeśli pusta, trzeba zmienić `bad_IRQNR_interrupt` na `[fast_]IRQNR_interrupt`) można dokonać przy pomocy funkcji `request_irq()`. Zaniechanie obsługi dokonuje się przy pomocy funkcji `free_irq()`.

```
int request_irq( unsigned int nr_irq, void (*funkcja_obsługi)(int,void*,struct pt_regs*),
                unsigned long flags, const char *nazwa_urzadzenia, void *dev_id_PCI);

free_irq( unsigned int irq_nr );
```

Podobnie jak w QNX, możliwe jest ustalenie kilku funkcji do obsługi przerwania (`flags` ustawione na `SA_SHIRQ`). Wówczas należy dokonać filtracji źródła przerwania przy pomocy funkcji `probe_irq_on()/probe_irq_of()`.

```
unsigned long probe_irq_on(void);          // podaje maskę bitową przerw z zarejestrowanymi procedurami obsługi
int probe_irq_of(unsigned long);          // używa owej maski podając nr przerwania, które wystąpiło
```

Własna obsługa przerwania, jeśli jest czasochłonna, powinna dzielić się na dwie części:

- część górną – ograniczoną do minimum, czytującą np. stan sprzętu, przekazującą dalszą obsługę do części dolnej,
- część dolną – wykonującą pracochłonne operacje na danych, związanych z obsługą przerwania.

Przekazanie wykonania z części górnej do dolnej można dokonać poprzez tablicę kolejki zadań `bh_base`, w której zamieszczone są funkcje zaznaczone (`mark_bh()`) do wykonania po zakończeniu obsługi przerwania a przed rozpoczęciem wykonywania pozostałych zadań.

Dostęp do sprzętu w systemie Windows NT

Sprzęg systemowy

W Windows NT dostęp do sprzętu jest w pełni zarządzany przez system. Ideą NT jest mikrojądro niezależne od sprzętu. Wyróżniając warstwy w systemie w kontekście niniejszego opracowania należy szczególną uwagę objąć HAL (*Hardware Abstraction Layer*). Dostęp do zasobów sprzętowych odbywa się jedynie poprzez sterowniki w trybie kernela (dostarczane jako pliki `*.sys`). Sterowniki te mają zaimplementowany zestaw niezbędnych funkcji, z których znamienne są:

- `DriverEntry()` – wywoływana podczas inicjalizacji sterownika przez system,
- `AddDevice()` – inicjalizacja sprzętu,
- `Unload()` – zakończenie współpracy ze sprzętem i usunięcie sterownika,
- `InterruptServiceRoutine()` – własna procedura obsługi przerwania, *etc.*

Na uwagę zasługuje technologia PnP (*Plug and Play*). Do jej zadań należy: wykrywanie sprzętu, przydzielanie mu odpowiednich sterowników, przydzielanie sterownikom zasobów sprzętowych wedle dostarczonych i realizowalnych konfiguracji (może dokonać rekonfiguracji, jeśli to konieczne). Kolejność uruchamiania jest następująca:

- inicjacja i uruchomienie sterowników niezgodnych z PnP (sztywne adresy zasobów),
- inicjacja sterowników PnP,
- przydział zasobów sterownikom PnP,
- uruchomienie sterowników PnP.

Dostęp do sterownika odbywa się poprzez funkcje systemowe SCM:

```
ControlService(); OpenService(); CreateService(); StartService(); DeleteService();
CloseServiceHandle();
```

Operacje na danych wykonuje się poprzez funkcję `DeviceIoControl()`; możliwy jest także, podobnie jak w systemach unixowych, dostęp strumieniowy poprzez funkcje `ReadFile()` / `WriteFile()` - nazwa urządzenia w konwencji UNC.

W systemie Windows NT wprowadzono asynchroniczne wywołania funkcji sterowników. Polegają one na ominięciu kaskadowego wywoływania powiązanych ze sobą sterowników, co obciąża znacznie stos i wydłuża czas operacji. W to miejsce stworzono komunikację poprzez komunikaty (pakiety) IRP (*Interrupt Request Packet*).

Dostęp do portów wejścia / wyjścia

Dostęp do wybranego obszaru wejścia/wyjścia należy zarezerwować. W tym celu ów obszar należy opisać w strukturze `IO_RESOURCE_DESCRIPTOR`, a następnie wpisać strukturę na listę zasobów sterownika `IO_RESOURCE_LIST`. Lista taka stanowi zestaw wymagań dla proponowanej konfiguracji. Wszystkie możliwe konfiguracje (czasem tylko 1) są wpisane do nadrzędnej listy `IO_RESOURCE_REQUIREMENTS_LIST`. Odczytu/zapisu z portu dokonuje się poprzez komunikaty IRP lub wywołując funkcję `DeviceIoControl()` (podając przesunięcie względem początku zarezerwowanego obszaru).

Obsługa przerwania sprzętowych

Obsługa przerwania sprzętowych jest możliwa jedynie przez sterowniki najniższego poziomu. Konieczna jest implementacja uruchamianej w trybie kernela funkcji `InterruptServiceRoutine()` (ISR). Funkcja ta ma być możliwie krótko wykonywana, aby nie blokować zgłoszeń przerwania równego lub niższego priorytetu (niezbędne odczyty stanu urządzenia itp.). Przetwarzanie danych należy przerzucić do niższego poziomu uprzywilejowania, wykorzystując funkcję DPC (*Deferred Procedure Call*). Wykonanie obsługi czasochłonnego przetwarzania danych może zostać przerzucone do funkcji `DpcForIsr()`, sterownik może też użyć jedną lub więcej procedur typu `CustomDpc()`. Procedury te należy zarejestrować; ich wywołanie odbywa się w postaci zamówienia na wykonanie (wstawienia do kolejki do wykonania) z jednoczesnym przekazaniem niezbędnych danych (kontekstu urządzenia).

```
typedef struct _IO_RESOURCE_DESCRIPTOR
{
    UCHAR Option;
    UCHAR Type;
    UCHAR ShareDisposition;
    UCHAR Spare1;
    USHORT Flags;
    USHORT Spare2;
    union {
        struct {
            ULONG Length;
            ULONG Alignment;
            PHYSICAL_ADDRESS MinimumAddress;
            PHYSICAL_ADDRESS MaximumAddress;
        } Port;
        struct {
            ULONG Length;
            ULONG Alignment;
            PHYSICAL_ADDRESS MinimumAddress;
            PHYSICAL_ADDRESS MaximumAddress;
        } Memory;
        struct {
            ULONG MinimumVector;
            ULONG MaximumVector;
        } Interrupt;
        struct {
            ULONG MinimumChannel;
            ULONG MaximumChannel;
        } Dma;
        struct {
            ULONG Length;
            ULONG Alignment;
            PHYSICAL_ADDRESS MinimumAddress;
            PHYSICAL_ADDRESS MaximumAddress;
        } Generic;
        struct {
            ULONG Data[3];
        } DevicePrivate;
        struct {
            ULONG Length;
            ULONG MinBusNumber;
            ULONG MaxBusNumber;
            ULONG Reserved;
        } BusNumber;
        struct {
            ULONG Priority;
            ULONG Reserved1;
            ULONG Reserved2;
        } ConfigData;
    } u;
} IO_RESOURCE_DESCRIPTOR,
*PIO_RESOURCE_DESCRIPTOR;
```

7. Pamięci masowe

- magazynowanie dużej ilości danych
- możliwość zapisu / odczytu
- podstawowe parametry: czas dostępu, prędkość odczytu, prędkość zapisu - mniejsze od parametrów pamięci operacyjnej, im krótsze, tym lepsze

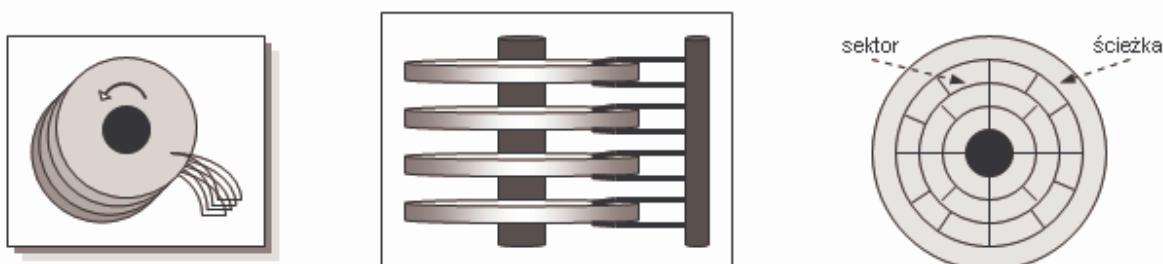
Karty perforowane, taśmy magnetyczne, dyski magnetyczne (FDD 5,25'', 3,5'', ZIP...), dyski twarde HDD, ...

7.1 Struktura dysków twardych (HDD)

Sektor - najmniejsza jednostka zapisanych danych na dysku.

Ścieżka - zbiór sektorów.

Cylinder - tworzą ścieżki o tym samym numerze na wszystkich talerzach (dostępne bez przemieszczania głowicy).



Macierze dysków RAID

- RAID 0 - przeplatanie danych między N dysków - N-krotne zwiększenie pojemności, zwiększone prawdopodobieństwo awarii,
- RAID 1 - zapis tych samych danych na N dyskach - łączna pojemność: 1 dysk, prawdopodobieństwo awarii N-krotnie mniejsze,
- RAID 2,3,4,5,6,... 0/1, 1/0

Planowanie dostępu do dysku (ruch głowicy)

Algorytm:

- FCFS – obsługa wg kolejności zgłoszeń – sprawiedliwa, obsługa nie jest najszybsza.
- SSTF – odpowiednik SJF – następny odczyt danych znajdujących się najbliżej – może prowadzić do zagłodzenia,
- SCAN – realizacja odczytów podczas przesuwania się ramienia od pierwszego do ostatniego cylindra a następnie w drugą stronę,
- C-SCAN – realizacja odczytów podczas przesuwania się ramienia od pierwszego do ostatniego cylindra (jeden kierunek głowicy),
- LOOK/C-LOOK – modyfikacja SCAN/C-SCAN – zakres przesuwania głowicy do najdalej położonego zamówienia.

7.2 Systemy plików

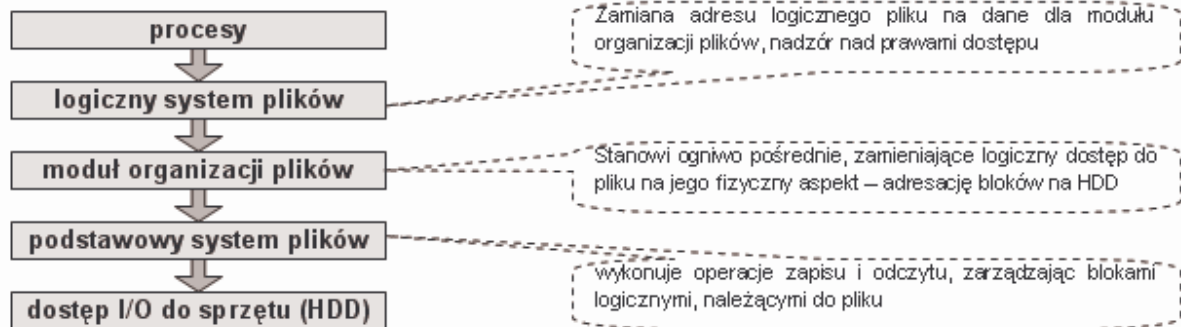
Wymagania:

- przechowywanie plików o różnych rozmiarach (0 - max B),
- organizacja drzewa plików / katalogów,
- realizacja adresacji,
- ochrona dostępu do plików / katalogów,
- szyfrowanie plików / katalogów,
- podstawowe operacje na plikach / katalogach.

Operacje na plikach / katalogach

- tworzenie, usuwanie, przesuwanie, zmiana nazwy, zmiana atrybutów, zmiana właściciela, zapis do pliku (od początku, dopisywanie, skracanie), odczyt z pliku, odczyt zawartości katalogu,

Organizacja systemu plików



7.3 Alokacja plików na dysku

- ciągła – każdy plik zajmuje kolejne bloki na dysku (np. ISO9600 – CD ROM)
 - + szybki dostęp – rzadka konieczność ruchu głowicy,
 - niemożność łatwego zwiększania rozmiaru pliku,
 - problem przydziału miejsca – pierwsze, najlepsze i najgorsze dopasowanie, przepisywanie zawartości dysku,
 - fragmentacja zewnętrzna,
 - fragmentacja wewnętrzna.
- łańcuchowa – plik jest listą kolejnych bloków na dysku, w tablicy kat. dostępny jest adres pierwszego (+ew. ostatniego) bloku
 - + łatwe i wydajne zarządzanie przestrzenią dyskową (brak fragmentacji zewnętrznej),
 - fragmentacja wewnętrzna,
 - sekwencyjny dostęp do pliku – większą wydajność osiąga się łącząc kolejne sektory w tzw. klaster,
 - awaria jednego bloku może pociągnąć destrukcję całego systemu plików.
- indeksowana - plik posiada własną listę adresów kolejnych lub wybranych bloków
 - + łatwe i wydajne zarządzanie przestrzenią dyskową (brak fragmentacji zewnętrznej),
 - + szybki dostęp do wybranego miejsca w pliku,
 - fragmentacja wewnętrzna.

Realizacje alokacji indeksowanej:

- łańcuchowa – plik posiada pewien łańcuch bloków, zawierających adresy bloków danych na dysku,
 - + szybki dostęp w przypadku małych plików,
 - + nieograniczona długość pliku
 - spowolnienie odczytu swobodnego w przypadku dużych plików,
 - lista bloków zajmuje dodatkowe miejsce,
- wielopoziomowa – dla pliku tworzone jest drzewo N-poziomowe, którego węzły zawierają adresy bloków dla odpowiedniego poziomu; liście w drzewie to bloki z danymi
 - większy niż w przypadku metody łańcuchowej narzut czasowy w przypadku małych plików,
 - + nieograniczona długość pliku
 - + jednakowy czas odczytu swobodnego w przypadku małych i dużych plików,
 - drzewo bloków zajmuje dodatkowe miejsce,
- kombinowana – połączenie metody wielopoziomowej i łańcuchowej, dopasowane statystycznie do charakterystyki danych.

Defragmentacja – łączenie klastrów zawierających kolejne części pliku w ciąg, celem przyspieszenia odczytu danych.

7.4 System FAT

MBR	DPT	BR	FAT	FAT*	RD	DATA.....
-----	-----	----	-----	------	----	----------------

MBR – Master Boot Record – obszar, do którego odwołuje się komputer podczas inicjacji, zawiera program do poprawnego odczytu DPT,

DPT – Disk Partition Table – opisy partycji dysku,

BR – Boot Record – obszar startowy danej partycji, uruchamiany, jeśli ona jest partycją główną. Zawiera informacje min. nt. tablic FAT, liczby bajtów w sektorze, liczby sektorów w klastrze,

FAT – File Allocation Table – tablica alokacji plików, zawiera dane nt. rozmieszczenia plików na dysku,

FAT* – kopia bezpieczeństwa tablicy FAT,

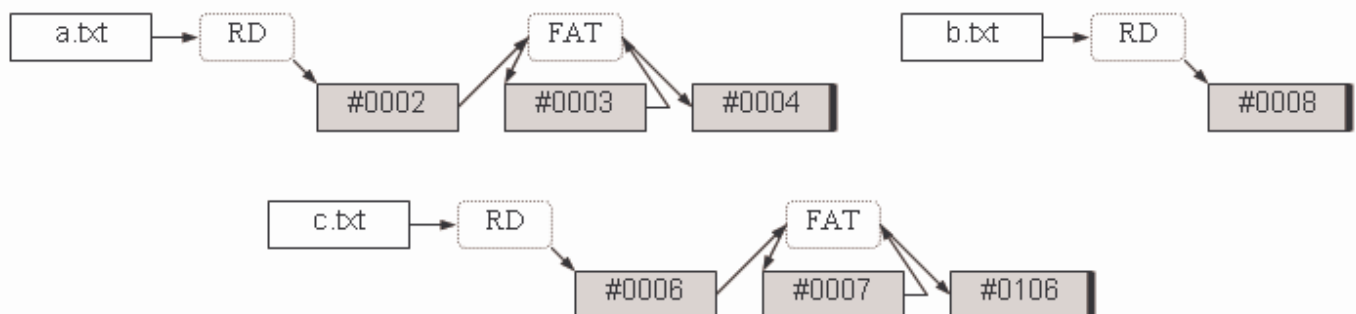
RD – Root Directory – dane katalogu głównego – max 512 wpisów (plików lub katalogów),

DATA – obszar danych.

FAT zawiera informacje o łańcuchach klastrów na dysku, zawierających dane plików. Adres startowy pliku (klaster) odczytuje się z RD. Następnie w tablicy FAT odczytywane jest, czy to ostatni klaster (koniec pliku) czy też wskazywany jest adres następnego klastra, zawierającego dane pliku. W FAT dany klaster może być również oznaczony jako wolny lub uszkodzony.

Przykładowa tabela FAT16

FAT	01	02	03	04	05	06	07	08	09	0A	0B	0C	...
00	ID	0003	0004	FFFF	0	0007	0106	FFFF					
01						FFFF							
..													



FAT 12 – dla małych dysków, np. FDD, HDD do 16MB, adresacja klastrów 12 bitami (000-FFF)

FAT 16 – adresacja klastrów 16 bitami (0000-FFFF)

FAT 32 – adresacja klastrów 32 bitami (00000000-FFFFFFFF)

Zalety:

- szybki dostęp do plików,
- efektywne magazynowanie danych dla małych dysków,
- liczne narzędzia i oprogramowanie.

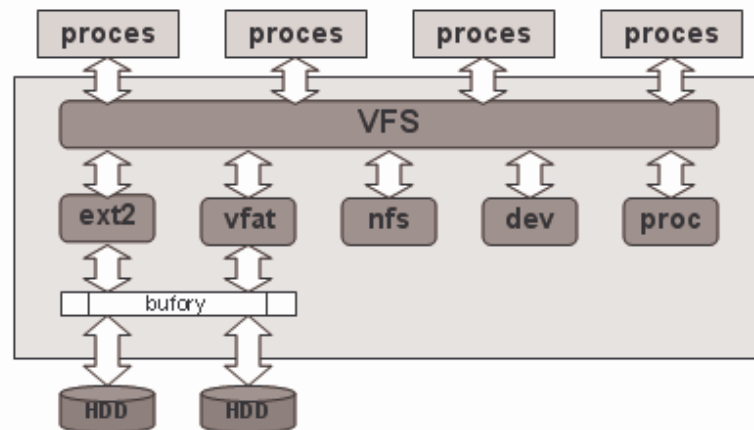
Wady:

- ograniczenie liczby wpisów katalogu głównego (512),
- ograniczona liczba klastrów (FAT16 – 65524),
- przeszukiwanie w drzewie katalogów – listowe,
- nieefektywne wykorzystanie miejsca dla dużych partycji – znaczący narzut w przypadku małych plików,
- brak podstawowej i zaawansowanej ochrony dostępu, brak kompresji.

Rozmiar partycji	FAT16	FAT32
16-32 MB	512B	
-64MB	1KB	512B
-128MB	2KB	1KB
-256MB	4KB	2KB
-512MB	8KB	4KB
-1GB	16KB	4KB
-2GB	32KB	4KB
-4GB	64KB	4KB
-8GB	-	4KB
-16GB	-	8KB

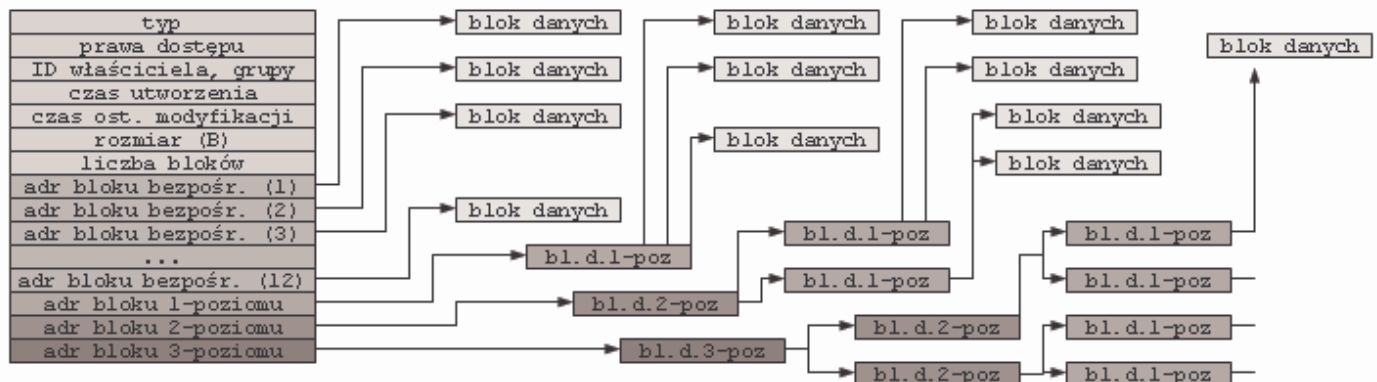
7.5 EXT2 (Linux)

VFS - Wirtualny system plików w Linuksie



Z każdym plikiem związana jest struktura i-węzła (i-node). Są przechowywane na dysku (zawierają adres swojej lokacji na HDD) oraz w pamięci, celem przyspieszenia operacji (używa się tu listy dwukierunkowej oraz tablicy mieszającej). Rozmiar bloku danych – 4KB. Kombinowana struktura drzewiasta pozwala na zapis plików o max dł. 16GB, ograniczonych do 4GB przez pole rozmiaru (4B).

Struktura i-węzła



Struktura partycji ext2



- blok startowy – zawiera informacje potrzebne podczas uruchamiania systemu,
- grupy bloków (8-128MB, ostatnia):
 - superblok – dane nt. systemu plików, sprzętu, rozmiar bloku, struktury do synchronizacji dostępu, etc.
 - deskryptory grupy – informacje nt. grup bloków, zarządzanie wolnym miejscem w grupie,
 - bitmapa bloku
 - bitmapa struktury i-node
 - tablica i-node
 - bloki danych

Odszukanie i-node danego pliku w grupie odbywa się przy następujących kalkulacjach:

$$\text{nr_bloku} = [(\text{nr_inode}-1) / \text{liczba_inode}] + \text{nr_bloku_zaw_tablice_inode}$$

$$\text{offset_w_bloku} = ((\text{nr_inode}-1) \% \text{liczba_inode}) * \text{sizeof}(i_nodehdd)$$

7.6 NTFS

- teoretycznie duża przestrzeń adresowa: 2⁶⁴-1 klastrow po 64KB, tabela partycji MBR obsługuje dyski do 2TB,
- transakcyjny system plików – operacje na danych wykonywane są w transakcjach, zachowujących spójność danych i pozwalających na ich odzyskiwanie w przypadku awarii (podoperacje są logowane), możliwe cofnięcie ostatniej operacji
- przeszukiwanie w drzewie katalogów – B+-drzewa,
- rozbudowane prawa dostępu – 13 typów uprawnień dla każdego z zestawu użytkowników/grup użytkowników
- kompresja plików, obsługa plików rozrzedzonych, quota,
- szyfrowanie plików EFS (klucz RSA 1024b),

Struktura partycji NTFS



MFT – Master File Table – organizowana jest poprzez metapliki (wybór):

0. \$ Mft – zawiera rekordy każdego pliku
1. \$MftMirr – kopia bezpieczeństwa \$Mft
2. \$LogFile – dziennik kolejnych operacji dyskowych, tworzących transakcje, pozwala na cofnięcie ostatniej operacji,
3. \$Volume – informacje o partycji,
5. \$. – dane katalogu głównego,
6. \$Bitmap – mapa zajętości klastrów,
8. \$BadClus – lista uszkodzonych klastrów.

Rekord MFT

W MFT zawarte są rekordy każdego pliku/katalogu partycji.

Inf. stand	nazwa	descr. bezpiecz	...dane...
------------	-------	-----------------	------------

- katalogi przechowują własne kopie rekordów MFT dla plików,
- obszar MFT nie powinien być fragmentowany – rezerwacja 12,5% pojemności partycji,
- rozmiar MFT nie jest ograniczony, w zależności od struktury plików/katalogów miejsce rezerwowe mogą zająć dane,
- małe pliki / katalogi (np. do 1500B, zależy od wielkości klastra) są w całości zapisywane w MFT (rezydentny),
- w przypadku większych plików lub informacji (np. fragmentacja...) do MFT wpisuje się tylko rekord bazowy (lista).

Logical Cluster Numbers - LCN - numery kolejnych klastrów na dysku.

Virtual Cluster Numbers - VCN - numery kolejnych klastrów w strumieniu pliku.

W rekordzie MFT, będącym listą, wpisuje się informacje o klastrach, zawierających dane strumienia pliku.

