AGH

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Systemy operacyjne
Wykład 02N

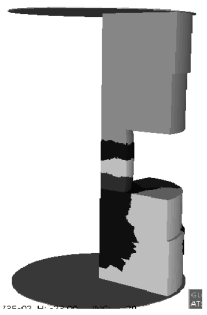
Wersja 2024

dr inż. Marek Wilkus http://home.agh.edu.pl/~mwilkus
Wydział Inżynierii Metali i Informatyki Przemysłowej
AGH Kraków

Na podstawie programu opracowanego przez dr inż. Krzysztofa Wilka

Procesy współpracujące

- Procesy są współpracujące, jeżeli jeden proces może wpływać na inne procesy, a inne procesy mogą wpływać na niego.



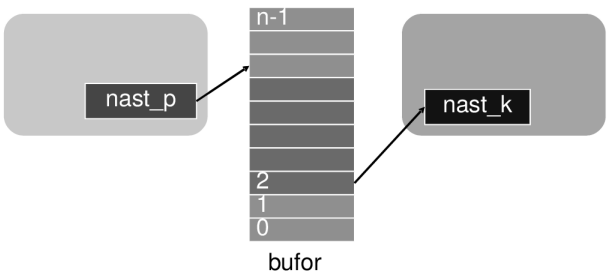
Symulacja numeryczna na 4 procesory. Jeden kolor - jeden proces.

Procesy współpracujące

- Zalety:
  - Dzielenie informacji – kilka procesów może korzystać z np. jednego pliku.
  - Przyspieszenie obliczeń w systemach wieloprocesorowych, gdy można podzielić zadanie na wykonywane równolegle mniejsze.
  - Modularność rozwiązań,
  - Wygoda – jeden użytkownik może wykonywać w tym samym czasie kilka zadań, np. edycję, kompilację, podgląd, drukowanie.

Komunikacja międzyprocesowa: Pamięć współdzielona

Współpraca – problem producenta i konsumenta



Bufor ograniczony – wspólne zmienne

```
var n;
type jednostka = …;
jednostka bufor[n-1];
jednostka* we,wy;
```

Następne wolne miejsce

Pierwsze zajęte miejsce

Proces producenta

```
while true {
  ...
   produkuj jednostka do nast_p;
...

   while ((we+1)%n==wy)  {} //nie produkuj
   bufor[we]=nast_p;
   we=(we+1)%n;
}
```


Proces konsumenta

```
while true {
     while (we==wy) {} //pusto
    nast_k = bufor[wy];
    wy=(wy+1)%n;
  ...
   przetwarzaj (nast_k);
  ...
}
```


Komunikacja międzyprocesowa: System komunikatów

Komunikacja międzyprocesowa

- Dwie operacje:
  - Nadaj komunikat,
  - Odbierz komunikat.
- W celu ich realizacji procesy:
  - Ustalają łącze komunikacyjne
  - Nadają i odbierają (również kodują i dekodują) komunikaty.
- Komunikacja odbywa się za pomocą pamięci dzielonej lub szyny systemowej.

Komunikacja międzyprocesowa – podstawowe kwestie:

- Jak ustanawia się połączenie?
- Czy jedno łącze może obsłużyć >2 procesy?
- Ile łączy pomiędzy parą procesów?
- Jaka przepustowość łącza? Obszar buforowy?
- Komunikaty: stałej czy zmiennej długości?
- Łącze jedno czy dwukierunkowe?
- Komunikacja: Bezpośrednia czy pośrednia?

Komunikacja bezpośrednia

- Dwie operacje elementarne:
  - nadaj(PID1, komunikat),
  - odbierz(PID2, komunikat).

(PID1, PID2 – identyfikatory procesów)

Właściwości łącza

- Właściwości łącza:
  - Ustanawiane automatycznie pomiędzy parą procesów, wystarczy, aby procesy znały swoje identyfikatory.
  - Dokładnie dla 2 procesów. Między parą dokładnie jedno łącze.
  - Zazwyczaj dwukierunkowe, dopuszczalne jednokierunkowe.

Łącza w problemie producenta i konsumenta

```
produkuj jednostka do nast_p;
nadaj(konsument, nast_p);

odbierz(producent,nast_k);
przetwarzaj(nast_k);
```


Komunikacja pośrednia

- Komunikaty nadawane i odbierane są za pomocą portów ("skrzynek pocztowych"). Procesy mogą się komunikować jedynie mając wspólny port.
- Łącze jest ustanawiane gdy procesy dzielą port.
- Łącze może być związane z więcej niż dwoma procesami.
- Każda para procesów może mieć kilka łączy, przez różne porty.
- Łącze może być jedno i dwukierunkowe.

Komunikacja pośrednia Problem trzech procesów

- Trzy procesy P1, P2, P3 dzielą wspólny port. Proces P1 wysyła komunikat, procesy P2 i P3 próbują go odebrać – powstaje konflikt.
- Rozwiązania:
  - Zezwalać jedynie na łącza pomiędzy dwoma procesami,
  - Zezwalać co najwyżej jednemu procesowi na wykonanie odbioru w danym momencie,
  - Dopuścić, by system wybrał proces docelowy. System powinien poinformować nadawcę o wyborze.

Buforowanie

- Kolejka komunikatów:
  - pojemność zerowa - łącze nie dopuszcza aby czekał w nim jakikolwiek komunikat - nadawca czeka aż odbiorca odbierze,
  - pojemność ograniczona - w kolejce może pozostawać tyle komunikatów, na ile zaprojektowano kolejkę. W przypadku kolejki pełnej nadawca musi czekać.
  - Pojemność nieograniczona - kolejka ma potencjalnie nieskończoną długość. Nadawca nigdy nie czeka.

Sytuacje awaryjne

- Zakończenie procesu - system musi rozwiązać problemy:
  - Gdy proces czeka na komunikaty z zakończonego,
  - Gdy nadaje komunikaty do zakończonego,
- Utrata komunikatów
  - System wykrywa to i ponownie nadaje komunikat,
  - Proces nadawczy wykrywa i ew. powtarza komunikat,
  - System wykrywa i powiadamia proces nadawczy.
- Zniekształcenie komunikatów - kontrola poprawności przez sumy kontrolne, sprawdzanie parzystości itd.

Komunikacja: Jądro Mach

- Większość wywołań i przepływ informacji odbywa się za pomocą komunikatów
- Przy tworzeniu zadania powstają dwa porty („skrzynki pocztowe”):
  - Jądra - komunikacja jądra z zadaniem,
  - Zawiadomień - wysyłanie informacji o zdarzeniach.
- Funkcje systemowe:
  - msg_send - wysyłanie komunikatu
  - msg_receive - odbieranie komunikatu
  - msg_rpc - wysyłanie blokujące do momentu odpowiedzi,
  - port_allocate - tworzenie nowego portu,

Gdy proces nie odpowiada...

- Skrzynka jest pełna. Proces nadający może:
  - Czekać na zwolnienie miejsca w skrzynce,
  - Czekać z limitem czasu,
  - Zignorować fakt i nie czekać,
  - Przekazać komunikat do systemu do późniejszego przesłania (może tak czekać tylko jeden komunikat.
- Podobny interfejs zastosowano w jądrze GNU/Hurd.

Komunikaty w Windows linii NT

- LPC - Local Procedure Call - zmodyfikowany mechanizm RPC do przesyłania komunikatów.
  Są dwa typy portów:
  - Łączące
  - Komunikacyjne.
- Komunikacja:
  - Klient uzyskuje uchwyt do obiektu portu,
  - Klient wysyła prośbę o połączenie (funkcja systemowa),
  - System/serwer tworzy dwa prywatne porty i przekazuje klientowi uchwyt do jednego z nich,
  - Klient i serwer wykorzystują uchwyty do przesyłu komunikatów.

Komunikacja w Windows linii NT (c.d.)

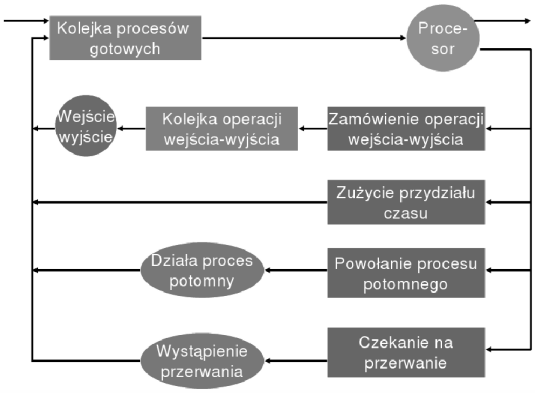
- Trzy sposoby przekazywania komunikatów:
  - Dla <256B - kolejka komunikatów jako pamięć tymczasowa, kopiowanie komunikatów z jednego procesu do drugiego.
  - Dla większych - pamięć dzielona (obiekt sekcji), by uniknąć kopiowania.
  - Dla np. funkcji graficznych - szybkie wywoływanie procedur lokalnych. Obiekt sekcji do przekazywania komunikatów, obiekt pary zdarzeń do synchronizacji.

Planowanie przydziału procesora

Kolejki planowania procesów

- Kolejka zadań (job queue) - tworzą ją procesy wchodzące do systemu.
- Kolejka procesów gotowych (ready queue) -procesy gotowe do działania, umieszczone w pamięci.
- Kolejki do urządzeń (device queue) - procesy czekające na konkretne urządzenie.

Diagram kolejek



Planiści

- Planista długoterminowy (planista zadań) - wybiera procesy **do kolejki procesów gotowych**, do pamięci.
  - Jest on wywoływany stosunkowo rzadko (sekundy) i nie musi być szybki.
- Planista krótkoterminowy (planista przydziału procesora) - wybiera proces z **puli procesów gotowych i przydziela mu procesor**.
  - Jest on wywoływany b. często (co ms) i musi być b. szybki.

Planiści

- Procesy możemy podzielić na:
  - Ograniczone przez wejście-wyjście (więcej czasu zajmują operacje we-wy niż korzystanie z procesora),
  - Ograniczone przez procesor (potrzebują znacznie więcej czasu procesora niż dla operacji we-wy).
- Zadaniem planisty długoterminowego jest dobór optymalnej mieszanki zadań ograniczonych przez procesor i przez I/O.

Planista średnioterminowy

- Występuje w niektórych systemach z podziałem czasu. Jego zadaniem jest, w koniecznych przypadkach, zmniejszanie stopnia wieloprogramowości poprzez wysyłanie części zadań chwilowo na dysk (swapping). Pomaga to w doborze lepszego zestawu procesów w danej chwili, lub dla zwolnienia obszaru pamięci.

Przełączanie kontekstu

- Podczas przejścia procesora z wykonywania jednego procesu do drugiego należy przechować stan starego procesu i załadować przechowany stan nowego.
- Z punktu widzenia systemu są to działania nieproduktywne, tak jak przygotowanie czy sprzątanie stanowiska pracy, ale są niezbędne przy wieloprogramowości.
- Mechanizm wątków pozwala na redukcję czasu przełączania kontekstu.

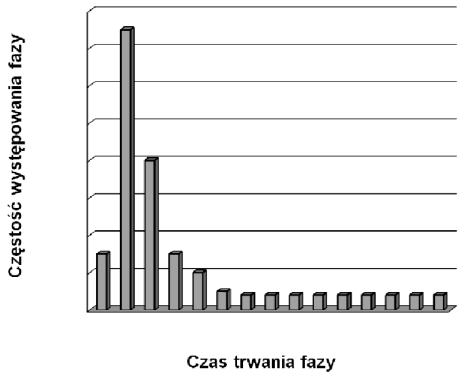
Planowanie przydziału procesora

- W pamięci operacyjnej znajduje się kilka procesów jednocześnie.
- Kiedy jakiś proces musi czekać, system operacyjny odbiera mu zasoby procesora i przekazuje do dyspozycji innego procesu.
- Planowanie przydziału procesora jest jedną z podstawowych funkcji każdego systemu operacyjnego.

Fazy procesora i I/O

załaduj **przechowaj** **dodaj** **przechowaj** **czytaj** z pliku	Faza procesora
Czekaj na urządzenie wejścia-wyjścia	Faza wejścia/wyjścia
przechowaj **dodaj** **zwiększ indeks** **pisz** do pliku	Faza procesora
Czekaj na urządzenie wejścia-wyjścia	Faza wejścia/wyjścia
załaduj **przechowaj** **dodaj** **przechowaj** **czytaj** z pliku	Faza procesora
Czekaj na urządzenie wejścia-wyjścia	Faza wejścia/wyjścia

Czasy faz procesora



Planowanie przydziału procesora

- Proces ograniczony przez I/O ma zazwyczaj dużo krótkich faz procesora.
- Proces ograniczony przez procesor może mieć mało, lecz bardzo długich faz procesora.

Planowanie przydziału procesora

- Decyzje o zmianie przydziału procesora podejmowane są gdy:
  - Proces przeszedł od stanu aktywności do czekania, np. na zakończenie potomka lub "zamówił" I/O.
  - Proces przeszedł od stanu aktywności do gotowości, np. wskutek wystąpienia przerwania.
  - Proces przeszedł od stanu czekania do gotowości, np. po zakończeniu operacji I/O.
  - Proces kończy działanie.
- Jeśli planowanie odbywa się tylko w pierwszym i ostatnim przypadku, mamy do czynienia z **planowaniem niewywłaszczeniowym (co-operative).** W przeciwnym wypadku planowanie jest **wywłaszczeniowe (pre-emptive)**.

Planowanie przydziału procesora

- Bez wywłaszczeń - proces, który otrzymał procesor, odda go dopiero przy przejściu do stanu oczekiwania lub po zakończeniu. Nie wymaga zegara. Stosowane w starszych systemach Windows.
- Planowanie wywłaszczające jest **ryzykowne:** Należy brać pod uwagę fakt, że proces w momencie wywłaszczenia może być w trakcie wykonywania funkcji systemowej.
- Zabezpieczenia:
  - System czeka z przełączeniem kontekstu do zakończenia funkcji,
  - Przerwania są blokowane przy przejściu do ryzykownych fragmentów kodu jądra,
  - Nie wywłaszcza procesu, gdy wewnętrzne struktury jądra są niespójne.

Dispatcher (ekspedytor)

- Jest to moduł, który przekazuje procesor do dyspozycji procesu wybranego przez planistę krótkoterminowego. Do jego zadań należy:
  - Przełączanie kontekstu,
  - Przełączanie procesu do trybu użytkownika,
  - Wykonanie skoku w kodzie procesu by wznowić jego działanie.
- Opóźnienie ekspedycji – czas jaki ekspedytor używa na wstrzymanie jednego procesu i wznowienie innego.
  Czas ten powinien być jak najkrótszy (ekspedytor jak najszybszy).

Kryteria planowania

- Wykorzystanie procesora - w realnych systemach od 40 (słabe) do 90% (intensywne),
- Przepustowość - mierzona ilością procesów kończonych w jednostce czasu (dla długich - kilka na godzinę, dla krótkich - kilka na sekundę),
- Czas cyklu przetwarzania - od nadejścia procesu do systemu, do jego zakończenia (suma czasów oczekiwania na wejście do pamięci, w kolejce procesów gotowych, okresów aktywności i operacji wejścia-wyjścia),
- Czas oczekiwania - suma czasów w których proces czeka w kolejce p. gotowych,
- Czas odpowiedzi - w systemach interakcyjnych - czas od wysłania żądania do rozpoczęcia odpowiedzi.

Planowanie optymalne

- **Maksymalne** wykorzystanie procesora,
- **Maksymalna** przepustowość,
- **Minimalny** czas cyklu przetwarzania,
- **Minimalny** czas oczekiwania,
- **Minimalny** czas odpowiedzi.

Metoda FCFS (pierwszy zgłoszony – pierwszy obsłużony)

Zgłaszają się 3 procesy:

1. P1	o czasie trwania fazy:	24 ms
2. P2	o czasie trwania fazy:	3 ms
3. P3	o czasie trwania fazy:	3 ms

Diagram Gantta dla planowania FCFS:

Średni czas oczekiwania wynosi więc:

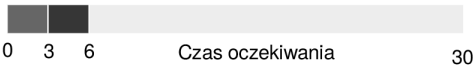
(0+24+27)/3 = 17 ms

Planowanie metodą FCFS

A gdyby procesy zgłosiły się w kolejności:

1. P2
2. P3
3. P1

Diagram Gantta dla planowania FCFS:



Średni czas oczekiwania wyniósłby wtedy:

(0+3 +6)/3 = 3 ms

Planowanie metodą FCFS

- Pierwszy przykład pokazuje, że planowanie metodą FCFS nie jest optymalne.
- Średni czas oczekiwania bardzo zależy od kolejności zgłoszenia procesów.
- Efekt konwoju - małe procesy czekają na zwolnienie procesora przez jeden wielki proces.
- Algorytm FCFS jest niewywłaszczający.
- Algorytm ten jest nieużyteczny w systemach z podziałem czasu, w których procesy powinny dostawać procesor w regularnych odstępach czasu.

Metoda SJF (Najpierw najkrótsze zadanie)

Zgłaszają się cztery procesy:

1. P1	o czasie trwania fazy:	6 ms
2. P2	o czasie trwania fazy:	8 ms
3. P3	o czasie trwania fazy:	7 ms
4. P4	o czasie trwania fazy:	3 ms

Diagram Gantta dla planowania SJF:

Średni czas oczekiwania wynosi:
(3+16+9+0)/4 = 7 ms

Dla metody FCFS wynosiłby:
(0+6+14+21)/4 =10,25 ms

Planowanie metodą SJF

- Można udowodnić, że planowanie tą metodą jest optymalne.
- Umieszczenie krótkiego procesu przed długim w większym stopniu zmniejsza czas oczekiwania krótkiego procesu niż zwiększa czas oczekiwania procesu długiego.
- Algorytm SJF jest często używany przy planowaniu długoterminowym.
- Problem - **nie można przewidzieć długości następnej fazy procesora**, można ją tylko szacować, najczęściej za pomocą średniej wykładniczej z poprzednich faz procesora.

Planowanie metodą SJF

$$f_{n+1} = \alpha\, t_n + (1-\alpha)\, f_n$$

gdzie:

α –współczynnik wagi (0÷1)

t_n _długość n-tej fazy procesora,

f_n - informacje o poprzednich fazach

gdy α =0 - bierzemy pod uwagę tylko dawniejszą historię,

a gdy α =1 - bierzemy pod uwagę tylko ostatnie fazy.

Najczęściej α =0,5

Planowanie metodą SJF

- Algorytm SJF może być wywłaszczający lub niewywłaszczający.
- Gdy do kolejki dochodzi nowy proces, który posiada fazę procesora krótszą niż pozostały czas w bieżącym procesie, algorytm wywłaszczający odbiera procesor bieżącemu procesowi i przekazuje go krótszemu.
- Metoda ta nazywa się **SRTF** (shortest remaining time first) czyli „najpierw najkrótszy pozostały czas".
- Algorytm niewywłaszczający pozwala na dokończenie fazy procesora.

Planowanie priorytetowe

- Mechanizm bardzo podobny do SJF, ale kryterium szeregowania jest priorytet procesu.
- Najpierw wykonywane są procesy o ważniejszym priorytecie.
- Priorytety mogą być definiowane wewnętrznie, na podstawie pewnych cech procesu (np. wielkość pamięci, limity czasu, zapotrzebowanie na urządzenia we-wy itd..)
- Priorytety definiowane zewnętrznie mogą np. zależeć od ważności użytkownika, jego firmy czy też od innych politycznych uwarunkowań.

Planowanie priorytetowe

- Wady:
  - Procesy o niskim (mało ważnym) priorytecie mogą nigdy nie dostać procesora (głodzenie, nieskończone blokowanie).
  - Przykład: w 1973 r. w wycofywanym z eksploatacji na MIT komputerze IBM 7094 wykryto „zagłodzony" proces o niskim priorytecie, który został uruchomiony do wykonania w 1967 roku ;-(.
- Rozwiązaniem problemu jest „postarzanie" czyli podnoszenie priorytetu procesów oczekujących zbyt długo.
- Przykład: przydział mieszkania dla dziadka w "Alternatywy 4".

Planowanie rotacyjne (RR) (Round Robin)

- Zaprojektowane specjalnie do systemów z podziałem czasu.
- Każdy proces otrzymuje kwant czasu (10-100ms), po upływie którego jest wywłaszczany i umieszczany na końcu kolejki zadań gotowych.
- Kolejny proces do wykonania jest wybierany zgodnie z algorytmem FCFS.
- Jeżeli jest **n** procesów gotowych a kwant czasu wynosi **q**, to każdy proces czeka nie dłużej niż (n-1)*q jednostek czasu.

Planowanie RR, kwant czasu 25ms

1. P1 o czasie trwania fazy: 24 ms
2. P2 o czasie trwania fazy: 3 ms
3. P3 o czasie trwania fazy: 3 ms

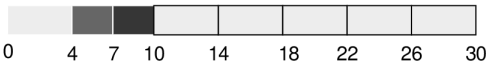
Diagram Gantta dla q=25 ms:

Średni czas oczekiwania wynosi:
(0+24+27)/3 = 17 ms
Przełączań kontekstu: 2

Planowanie RR, kwant czasu 4ms

1. P1 o czasie trwania fazy: 24 ms
2. P2 o czasie trwania fazy: 3 ms
3. P3 o czasie trwania fazy: 3 ms

Diagram Gantta dla q=4 ms:

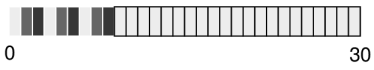


Średni czas oczekiwania wynosi:
(17)/3 = 5,66 ms
Przełączeń kontekstu: 7

Planowanie RR, kwant czasu 1ms

1. P1 o czasie trwania fazy: 24 ms
2. P2 o czasie trwania fazy: 3 ms
3. P3 o czasie trwania fazy: 3 ms

Diagram Gantta dla q=1 ms:



Przełączeń kontekstu: 29

Planowanie rotacyjne

- Wydajność algorytmu zależy bardzo od kwantu czasu **q**.
  - Gdy q jest duże, algorytm RR przechodzi praktycznie w algorytm FCFS.
  - Gdy q jest bardzo małe, to przez znaczną część czasu procesor zajęty jest przełączaniem kontekstu.
- Ogólna zasada:
  80% faz procesora powinno być krótszych niż kwant czasu.

Wielopoziomowe planowanie kolejek

- Kolejka procesów gotowych jest podzielona na oddzielne pod-kolejki, na przykład:
  - Kolejka procesów pierwszoplanowych (foreground),
  - Kolejka procesów drugoplanowych (background).
- Przykładowe proponowane algorytmy planowania:
  - Procesy pierwszoplanowe: RR
  - Procesy drugoplanowe: FCFS
- Procesy pierwszoplanowe mają absolutny priorytet nad drugoplanowymi.
- Aby nie „zagłodzić” procesów 2-planowych, stosuje się podział czasu na kolejki, przykładowo:
  - Kolejka pierwszoplanowa - 80% czasu procesora,
  - Kolejka drugoplanowa - pozostałe 20%

Kolejki wielopoziomowe ze sprzężeniem zwrotnym

- Mechanizm ten pozwala na przesuwanie procesów pomiędzy kolejkami.
- Proces, który używa za dużo procesora, można „karnie” przenieść do kolejki o niższym priorytecie i przez to dać szerszy dostęp do procesora innym procesom.
- Dzięki temu procesy ograniczone przez we-wy i procesy interakcyjne mogą pozostać w kolejkach o wyższych priorytetach.
- Długo oczekujące procesy z kolejki niskopriorytetowej mogą być przeniesione do ważniejszej - działa mechanizm postarzania procesów (przeciwdziała ich głodzeniu).
- **Planowanie ze sprzężeniem zwrotnym jest najbardziej złożonym algorytmem planowania przydziału procesora.**

Kolejki wielopoziomowe ze sprzężeniem zwrotnym - przykład

- Kolejka trzypoziomowa: K0, K1,K2
- Proces wchodzący trafia do kolejki k0 i dostaje kwant czasu 8 ms.
- Jeśli nie zakończy się w tym czasie, jest wyrzucany na koniec niższej kolejki K1.
- Gdy kolejka K0 się opróżni i przyjdzie czas wykonywania naszego procesu, dostaje on kwant czasu 16 ms.
- Jeśli i w tym czasie proces nie skończy działania, jest wyrzucany na koniec kolejki K2, obsługiwanej w porządku FCFS (oczywiście pod warunkiem, że kolejki K0 i K1 są puste).
- Tak więc najszybciej wykonywane są procesy do 8 ms, nieco wolniej procesy od 8 do 8+16=24 ms, a najdłużej czekają procesy długie (są obsługiwane w cyklach procesora nie wykorzystanych przez kolejki 1 i 2).

Planowanie w systemie Solaris

- Planowanie w wielopoziomowych kolejkach ze sprzężeniem zwrotnym,
- 4 klasy procesów: **real time, system, time sharing, interactive,**
- Priorytet globalny i priorytety w obrębie klas,
- Proces potomny dziedziczy klasę i priorytet,
- Klasa domyślna - **time sharing,**
- Im większy priorytet, tym mniejszy kwant czasu
- Klasa **system** - procesy jądra,
- Klasa **interactive** - wyższy priorytet mają aplikacje graficzne X11,
- Wątki o tym samym priorytecie planowane są algorytmem RR.

Planowanie w systemie Linux (od 2.6) [wcześniej był RR]

CFS - Completely Fair Scheduler - wykorzystuje jako kolejkę procesów gotowych **drzewiastą strukturę** (drzewa czerwono-czarne) szybkie przy wyszukiwaniu, ale O(log n) przy zapisie.

- Najpierw przydział dostają procesy o najniższym otrzymanym dotychczasowo czasie wirtualnym (vruntime), przechowywanym w nanosekundach.
- Jeżeli proces zakończył działanie, usuwany jest zupełnie z kolejki.
- Jeżeli proces osiągnął swój maksymalny czas działania lub został wstrzymany / oczekuje, jest umieszczany w drzewie-kolejce w miejsce określone za pomocą czasu przydziału zaktualizowanego o bieżąco wykorzystany. Następnie wybierany jest kolejny proces o najmniejszym czasie przydziału (co w drzewie ma niewielki koszt obliczeniowy).

CFS - vruntime

- vruntime określany jest z czasu działania danego procesu w stosunku od liczby aktualnie działających procesów o tym samym współczynniku redukcji.
- **Priorytety** - są współczynnikami redukującymi otrzymywany czas (niższy priorytet - bardziej zredukowany czas).
- Konfiguracja - dostosowanie dwóch opcji:
  - Latency - maksymalne dopuszczalne opóźnienie (mniejsze dla środowisk interakcyjnych, większe w węzłach obliczeniowych, serwerach)
  - Granularity - minimalny czas działania procesu, by system nie zajmował się tylko przełączaniem.
- Przy czym procesy wstrzymujące działanie na mniej niż swój kwant czasu mają czas proporcjonalny do tej wartości odejmowany od vruntime (nie blokować ciągłym przełączaniem kontekstu!).

CFS - ile trwa kwant? (obliczenia wewnętrzne jądra systemu)

- Każdy proces ma priorytet *statyczny* od 100 (najwyższy) do 139 (najniższy).
- Podstawowy kwant wyliczany jest ze wzoru:

$$\underset{\text{(in milliseconds)}}{\textit{base time quantum}} = \begin{cases} (140 - \textit{static priority}) \times 20 & \text{if } \textit{static priority} < 120 \\ (140 - \textit{static priority}) \times 5 & \text{if } \textit{static priority} \geq 120 \end{cases}$$

- Czyli wyższy priorytet → większy kwant.
- Każdy proces ma również priorytet *dynamiczny* używany przy wyborze z kolejki procesów gotowych:

$$\textit{dynamic priority} = \max(100, \min(\textit{static priority} - \textit{bonus} + 5, 139))$$

Bonus - 0..10, <5 zmniejsza priorytet, >5 podnosi go. Zależy od średniego czasu wstrzymywania procesu ze względu na I/O.

CFS - określanie granularity

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

CFS - proces interaktywny

- Proces zostaje uznany za „interaktywny” gdy spełnione jest:

$$dynamic\ priority \leq 3 \times static\ priority / 4 + 28$$

- Czyli:

$$bonus - 5 \geq \underbrace{static\ priority / 4 - 28}_{interactive\ delta}$$

- Bonus <5 powoduje, że proces nie „zajmie” czasu innym.
- Bonus ≥5 jest dla procesów o wyższym czasie aktywnego oczekiwania.
- Procesy interakcyjne są umieszczane zawsze w aktualnie planowanej kolejce.

Przełączenie kontekstu w CFS

1. Aktualizujemy licznik kwantu czasu.
2. Jeżeli proces wyczerpał swój czas, to musimy przełączyć proces:
3. Usuń proces z listy procesów aktywnych.
4. Zaznacz proces do ponownego przeliczenia czasu.
5. Zaktualizuj priorytet dynamiczny procesu.
6. Zresetuj licznik kwantu czasu.
7. Oznacz proces jako zakończony lub przełączony.
8. Wstaw wywłaszczany proces do listy aktywnych lub zakończonych.
9. PRZEŁĄCZ PROCESY (zapis bloku kontrolnego, przywołanie bloku, wznowienie).

Linux - specjalne procesy

- Możliwe jest zdowngrade’owanie metody planowania procesu. W ten sposób można naiwnie utworzyć klasy procesów.
- SCHED_FIFO - kolejka procesów planowanych metodą FIFO może uruchamiać się gdy kolejka innego planisty nie wymaga tego.
- SCHED_RR - Kolejka procesów planowanych RR.
- PREEMPT_RT (2024) – planuje procesy tak, by system zachowywał się **maksymalnie przewidywalnie**.

Planowanie w systemach Windows

- Priorytetowe z wywłaszczeniami,
- NT: 32 kolejki procesów, 6 klas priorytetów, 7 względnych priorytetów w obrębie klas,
- Priorytet domyślny w klasie: **normal**.
- Wątek jest wykonywany aż:
  - Zostanie wywłaszczony przez proces o wyższym priorytecie,
  - Zużyje swój kwant czasu,
  - Wykonuje operacje I/O blokujące zasoby lub wykorzystuje taki sterownik (tryby zgodności),
  - Zakończy się.
- Proces powiązany z aktywnym oknem ma zwiększany kwant czasu (NT5.1 – trzykrotnie).

Windows NT i późniejsze

- Rotacyjnie (Round-robin) z dodatkowymi priorytetami i wielopoziomową kolejką z informacją zwrotną.
- Niektóre procesy mają intencjonalnie zmniejszone priorytety (procesy z harmonogramu zadań, defragmentator dysków). Kryterium jest proces nadrzędny lub charakterystyka programu.
- Windows 10: Wielopoziomowe kolejki ze sprzężeniem zwrotnym.

Systemy mobilne

- Proces na pierwszym planie ma pierwszeństwo.
- Procesy odsyłane do tła mogą mieć czas "karny", przez który nie dostaną procesora.
- Procesy w tle mogą mieć ograniczony dostęp do API systemowego (np. tylko powiadomienia i sieć).

Systemy mobilne

- Przyjmując roboczo, że proces w pamięci stałej urządzenia mobilnego to niemal to samo co proces odswap-owany na dysk (niewielkie koszty czasowe załadowania procesu z SSD), w wielu systemach mobilnych **planista średnioterminowy** dodaje w miarę potrzeb ważniejsze usługi systemowe.
- Podejście to gwarantuje, że usługa systemowa potrzebująca akurat przetwarzać nadchodzące dane (łączność, I/O, telemetria) uruchomi się nawet, gdy jej proces nie działa.
- Stosowane jedynie dla nielicznych, krytycznych usług systemu.

Systemy mobilne

- Wiele programów działa w tle, a ich pełne zakończenie występuje rzadko – stąd planista powinien uwzględniać taką sytuację.
- Procesy te, gdy działają w tle, nie wymagają zupełnie planowania jak procesy interakcyjne – stąd Symbian przenosił je „z automatu" do kolejek o niższych priorytetach.
- Android/iOS dostosowują położenie tych procesów dynamicznie.

Planowanie w Systemie HaikuOS

- 3 możliwe strategie: simple, simple-SMT (dla procesorów wielordzeniowych), affine (wersja bieżąca).
- Wersja docelowa, po wersji beta, aktualnie w implementacji:
  - Dwie klasy procesów: Priorytet (p) 1-99 dostają $k*2^p$ kwant czasu, powyżej 99 są realtime i inne procesy mogą działać tylko, gdy te zwolnią CPU.
  - Proces może stać się „realtime" w specjalnych okolicznościach związanych z obsługą układów akceleratora i grafiki.
  - Aktywne okno ma większe szanse stać się „realtime".
  - Serwery i kity nie będące częścią jądra mogą stać się „realtime" używając funkcji jądra.

Synchronizacja procesów

- Proces producenta z wykorzystaniem licznika:

```
while (1) {
  ...
  produkuj (jednostka, do nast_p);
 ...

  while (licznik==n) do {} //oczekuj bez produkcji
 bufor[we]=nast_p;
 we=(we+1)%n;
 licznik++;

}
```


Synchronizacja procesów

- Proces konsumenta – z licznikiem:

```
while (1) {

  while (licznik==0) do {} //czekaj bez przetwarzania
  nast_k=bufor[wy];
  wy=(wy+1)%n;
  licznik--;
 ...
 przetwarzaj (jednostka z nast_k);
 ...

}
```


Synchronizacja procesów - przykład

- Wartość licznika wynosi 5.
  Po tym czasie producent wyprodukował 1 jednostkę, konsument przetworzył również 1 jednostkę.
- Ile wynosi licznik?

Synchronizacja procesów - przykład

1) P: rejestr1=licznik //r1=5
2) P: rejestr1++; //r1=6
3) K: rejestr2=licznik //r2=5
4) K: rejestr2--; //r2=4
5) P: licznik=rejestr1; //l=6
6) K: licznik=rejestr2 //l=4

Licznik wynosi 4

5') K: licznik=rejestr2 //l=4
6') P: licznik=rejestr1 //l=6

Licznik wynosi 6.

Bez synchronizacji możemy nigdy nie uzyskać 5.

Szkodliwa rywalizacja – race condition

- Jeżeli kilka procesów współbieżnie wykorzystuje i modyfikuje te same dane, to wynik działań może zależeć od kolejności w jakiej następował dostęp do danych. Następuje wtedy szkodliwa rywalizacja.

- **Sekcja krytyczna**

  Każdy ze współpracujących procesów posiada fragment kodu w którym następuje zmiana wspólnych danych. Jest to **sekcja krytyczna** procesu.

  Jednym z zadań synchronizacji jest zapewnienie sytuacji, w której gdy jeden proces jest w swojej sekcji krytycznej to inne nie mogą wówczas wejść do swoich sekcji krytycznych.

  Stąd każdy proces musi prosić (w sekcji wejściowej) o pozwolenie na wejście do swojej sekcji krytycznej.

Budowa procesu z sekcją krytyczną:

```
while (1) {

  ...

  sekcja wejściowa
      sekcja krytyczna
  sekcja wyjściowa

  ...

}
```


Warunki poprawnego działania sekcji krytycznej

- **Wzajemne wykluczanie**: Jeżeli proces działa w swojej sekcji krytycznej, to żaden inny proces nie działa w swojej (celem zachowania spójności).

- **Postęp**: Tylko procesy nie wykonujące swoich reszt (wszystkiego poza sekcją krytyczną i sekcjami w/w) mogą kandydować do wejścia do sekcji krytycznych i ten wybór nie może być odwlekany w nieskończoność (celem m.in. uniknięcia zbędnego blokowania).

- **Ograniczone czekanie**: Musi istnieć graniczna ilość wejść innych procesów do ich sekcji krytycznych po tym gdy jakiś proces zgłosił chęć wejścia do swojej i zanim otrzymał na to pozwolenie (celem nie zagłodzenia procesu).

Przykładowy algorytm synchronizacji

- Zmienne **wspólne**:
  int znacznik[2];
  int numer; //0..1
- Na początku znacznik[0]=znacznik[1]=0
- Odpowiedni fragment procesu *i* (proces konkurencyjny ma numer *j*):

```
while (1) {
 . . .
  znacznik[i]=1;
  numer=j;
  while (znacznik[j]==1 and numer==1) do {} //czekaj na wejście
        sekcja krytyczna
  znacznik[i]=0;
        reszta
 . . .
}
```


Rozkazy niepodzielne

- Są to sprzętowe rozkazy składające się z kilku kroków, ale muszą być wykonywane nieprzerwanie, np.:

```
bool testuj_i_ustal(bool* znak) {
      bool tmp=*znak;
      *znak=true;
      return tmp;
}
```

- Przykładowe użycie:

```
while (1) {
 . . .
while (testuj_i_ustal(wspolna)) do {}
      sekcja krytyczna
wspolna=false;

      reszta
 . . .
}
```

Rozkazy niepodzielne - po co?

- Pomiędzy wykryciem zwolnionego dostępu do sekcji krytycznej a wejściem nastąpiło przerwanie wstrzymujące wejście do sekcji.
- Inny proces też zauważa zwolnienie sekcji.
- Dwa procesy wchodzą jednocześnie do sekcji krytycznej.

- Systemy jednoprocesorowe: Blokujemy przerwania.
- Systemy wieloprocesorowe: Blokujemy przerwania, alokację rdzeni, przełączanie zadań na rdzenie… (wiele różnych algorytmów)

Semafory

- Są to sprzętowe zmienne całkowite, do których dostęp możliwy jest tylko przez dwie niepodzielne operacje:
  - czekaj(S): while (S<=0) {}
S--;
  - sygnalizuj(S): S++;

Czekaj, aż będzie można zdekrementować!

- Zastosowanie:

```
semafor wspolna;
while (1) {
 czekaj(wspolna);
   sekcja krytyczna
 sygnalizuj(wspolna);
   reszta
}
```


Przykład:

- Instrukcja S2 w procesie P2 musi być wykonana po zakończeniu wykonywania instrukcji S1 w procesie P1.

```
S1;
sygnalizuj(sync);


czekaj(sync);
S2;
```


Semafory z blokowaniem procesu

- Unika się "zapętlenia" procesu przed semaforem.
- Proces zamiast aktywnie czekać, umieszczany jest w kolejce związanej z danym semaforem i "usypiany".
- Operacja **sygnalizuj** wykonana przez inny proces "budzi" proces oczekujący i umieszcza go w kolejce procesów gotowych do wykonania.

```
type semaphore = record {
        int wartość;
        L[] list of processes;
}
```


Semafory z blokowaniem

```
czekaj(S):
        S.wartość--;
        if (S.wartość<0) {
                dołącz dany proces do listy;
                blokuj;
        }

sygnalizuj(S):
        S.wartość++;
        if (S.wartość<=0) {
                usuń proces  P z listy;
                obudź(P);
        }
```

Jeżeli wartość<0, to abs(wartość) – **liczba procesów czekających na ten semafor!**

Semafory a systemy wieloprocesorowe

- W systemach jednoprocesorowych niepodzielność operacji „czekaj” i „sygnalizuj” można zapewnić poprzez blokadę przerwań na czas wykonywania ich rozkazów.

- W środowisku wieloprocesorowym nie ma możliwości blokowania przerwań z innych procesorów - w takim przypadku wykorzystuje się rozwiązania z sekcji krytycznych - operacje „czekaj” i „sygnalizuj” są sekcjami krytycznymi. Ponieważ ich kody są małe (kilkanaście rozkazów), to zajmowane są rzadko i przypadki aktywnego czekania nie występują często i trwają krótko.

Semafory w UNIX

- man…
  - sem_overview
  - sem_wait
  - sem_post

Problem czytelników i pisarzy
(synchronizacja procesów zapisujących i czytających te same dane)

- Problem 1: Żaden z czytelników nie powinien czekać chyba, że pisarz w tym momencie pisze,
- Problem 2: Jeśli pisarz czeka na dostęp, to żaden nowy czytelnik nie rozpocznie czytania.

- Rozwiązanie:
  Procesy dzielą zmienne:
  semaphore wyklucz, pisanie;
  int liczba_czyt;

- Semafor **pisanie** jest wspólny dla czytających i piszących, obydwa na początku mają wartość 1, a liczba czytelników=0.
- Semafor **pisanie** organizuje wykluczanie piszących, jest również zmieniany przez pierwszego i ostatniego czytającego.

- Proces zapisujący:

```
czekaj(pisanie);
…
zapis
…
sygnalizuj(pisanie);
```

- Proces czytający:

```
czekaj(wyklucz); //tylko 1 proces może działać w tej sekcji!
liczba_czyt++;
if (liczba_czyt==1)
   { czekaj(pisanie); } //może być wewnątrz piszący!
sygnalizuj(wyklucz);  //mogą wchodzić inni czytający!
…
czytanie
czekaj(wyklucz); //znowu sekcja wyłączna
liczba_czyt--;
if (liczba_czyt==0)
   { sygnalizuj(pisanie); } //może ewentualnie wejść piszący
sygnalizuj(wyklucz);
```


Region krytyczny

- Jest to konstrukcja służąca do synchronizacji w językach wyższego poziomu.
- Przykładowa składnia:

  region V **when** B **do** S;

  gdzie:
  V – zmienna współdzielona

  Podczas wykonywania instrukcji **S** żaden inny proces nie ma prawa dostępu do zmiennej **V**.

  Jeśli warunek **B** jest prawdziwy, proces może wykonać instrukcję **S**, w przeciwnym wypadku czeka na zmianę **B** oraz na opuszczenie sekcji krytycznej przez inne procesy.

Synchronizacja w systemie Solaris

- Ze względu na implementację procesów czasu rzeczywistego, wielowątkowość i obsługę wielu procesorów, synchronizacja za pomocą sekcji krytycznych nie znalazła zastosowania.
- Zastosowano **zamki adaptacyjne**.

- Zamek rozpoczyna działalność jak standardowy semafor. Jeśli dane są już w użyciu, to zamek wykonuje jedną z dwu czynności:
  - Jeśli zamek jest utrzymywany przez wątek aktualnie wykonywany, to inny wątek ubiegający się o zamek będzie czekać (gdyż aktywny wątek niedługo się zakończy),
  - Jeśli zamek jest utrzymywany przez wątek nieaktywny, to wątek żądający zamka blokuje się i usypia, gdyż czekanie na zamek będzie dłuższe.

Synchronizacja w systemie Solaris

- Zamki adaptacyjne stosuje się, gdy dostęp do danych odbywa się za pomocą krótkich fragmentów kodu (zamknięcie na czas wykonywania co najwyżej kilkuset rozkazów).
- W przypadku dłuższych segmentów kodu stosuje się **zmienne warunkowe**.
- Jeśli zamek jest zablokowany, to wątek wykonuje operację **czekaj** i usypia. Wątek zwalniający zamek sygnalizuje to następnemu z kolejki uśpionych co tamtego budzi.
- **Blokowanie zasobów w celu pisania lub czytania** jest wydajniejsze niż używanie semaforów, ponieważ dane mogą być czytane przez kilka wątków równocześnie, a semafory dają tylko indywidualny dostęp do danych.

Synchronizacja w systemie Linux

- W nowszych wersjach jądra Linux występuje mechanizm synchronizacji zwany **futex** (Fast Userspace muTEX)**.**
- Proces w przestrzeni użytkownika widzi futex jako zmienną (z reguły 32-bit) dostępną przez szybkie, niepodzielne operacje nie wymagające długotrwałych funkcji systemowych podczas czekania.
- Funkcje systemowe używane są gdy rezerwujemy dostęp. Wątek, który potrzebuje dostępu do sekcji krytycznej używa **WAIT(adres,wartosc)**. Taki wątek jest zasypiany jeżeli wartość futexu spod adresu addr jest równa val.
- Gdy inny wątek wykona właściwą pracę, woła **WAKE(adres,wartosc)**. Wówczas wartość jest dekrementowana a jądro przesuwa czekające wątki do kolejki procesów gotowych by kontynuowały pracę.

Futex i więcej wątków

- A co gdy czeka więcej wątków?
- Zjawisko „**owczego pędu”** (thundering herd) - po zwolnieniu zasobów wybudzane jest wiele, wiele wątków i nie wiemy który pierwszy wejdzie do swej sekcji krytycznej.
- Jądro systemu obsługuje **kolejki oczekujących wątków.** Można realizować taką czynność wykorzystując specyficzne wartości **val**. Albo użyć funkcji przenoszących wątki do innych futeksów.

- **REQUEUE -** przyjmuje dwa futeksy i ilość wątków. Jeżeli wartość przechowywana w futeksie jest równa zadanej, budzi podają liczbę wątków, a zadaną część pozostałych przesuwa do nowego futeksu minimalizując zjawisko wybudzania dużej ilości wątków jednocześnie.
- **WAKE_OP -** Budzi wątki w zależności od wyniku zadanej w argumencie operacji. Użyteczne do implementacji zależnych od warunku technik synchronizacji.

Futex - krótka historia

- W Linuksie obecne od dawna, lecz API nie było stabilne.
- Futex2 - zaproponowane przez Valve - możliwości separowania zadań, różne długości zmiennej, warunkowe budzenie procesów.
- W Linuksie stabilne od wersji ok. 5.16 (jak wszystko dobrze pójdzie) - 2022 r.

Bariera pamięci

- Operacje dostępu do pamięci pojawiają się z różnych rdzeni i wątków, a również z urządzeń I/O (system DMA).

- A co jeżeli zmiana kolejności spowoduje szkodliwą rywalizację? Czy stosować kompletne semafory, futeksy, regiony krytyczne?

- Potrzebny jest niskopoziomowy mechanizm, który wymusi kolejność wykonywania operacji - zablokuje procesy zrównoleglonych zadań do wystąpienia zdarzenia (np. dotarxcia wszystkich wątków do tego momentu).

  Takim rozwiązaniem jest **bariera pamięci**.
- Wymusza konkretną kolejność wykonywanych operacji.
- Dzięki temu przy wielu procesorach nie dojdzie do szkodliwej rywalizacji nawet gdy optymalizacja dostępu do pamięci czy po prostu zmiana obciążenia rdzeni próbowałaby zmienić kolejność.
- We współczesnych systemach jest niskopoziomową instrukcją procesora.

Konfiguracje barier i ich zastosowania

- **Bariera zapisu** - daje pewność, że operacje zapisu przed barierą zostaną wykonane przed operacjami po wystąpieniu instrukcji bariery.
  - Przydatne gdy któryś wątek uparcie buforuje wartość zmiennej.

- **Bariera zależności danych** - Jeżeli popełniamy dwa odczyty, a drugi zależy od wyniku pierwszego (np. drugi korzysta z adresu w pamięci zaczytanego w wyniku pierwszego odczytu), to zastosowanie tej bariery wymusza aktualizację czytanej wartości (np. z innych wątków). Inne wątki nie odłożą w czasie aktualizacji pierwszej wartości.

- **Bariera odczytu** - Jak wyżej, ale dodatkowo gwarantuje, że wszystkie odczyty sprzed bariery zostaną wykonane przed odczytami po jej użyciu.

- **Bariera ogólnego przeznaczenia** - działa zarówno na operacje odczytu i zapisu - to, co przed barierą wykona się najpierw. Stosowana przy obliczeniach równoległych.

Kiedy stosujemy bariery?

- Kiedy istnieje możliwość, że pojawią się niepożądane interakcje przy pracy dwóch wątków, lub wątków i urządzenia wejścia-wyjścia, nad jednym miejscem w pamięci.

- Kiedy przerwanie (np. to od przełączania kontekstu) spowoduje zmianę alokacji rdzenia przy danym procesie - bariera pamięci umożliwia podjęcie pracy przez drugi rdzeń na aktualnych danych z pamięci. Bez niej aktualne dane pozostałyby w cache lub rejestrach rdzenia, któremu odebrano zadanie.
  - Taki błąd powodował, że w konsoli Nintendo Switch oprogramowanie sprzed wersji 14.0.0 powodowało w niektórych grach "glitche". Komunikacja z GPU odbywa się w dużej mierze z użyciem cache.