Opracowanie zerówka 2017

Koźlak van Gogh

by MajronMan

Wykład 1 - Wprowadzenie

Zadania Systemów Operacyjnych. Różne spojrzenia na Systemy Operacyjne

System operacyjny ma ukrywać złożoność operacji na sprzęcie i dostarczać wygodniejszy zbiór instrukcji. SO jest częścią oprogramowania, które jest uruchamiane w trybie jądra/trybie użytkownika. Na system operacyjny można spojrzeć z wielu perspektyw:

- **Przeciętny użytkownik** komputera powie, że system operacyjny jest tym, co dostawca przesyła w odpowiedzi na nasze zamówienie “systemu operacyjnego”.
- **Anegdota** Koźlaka: system operacyjny – przypomina rząd: Nie wykonuje sam żadnej użytecznej funkcji, ale ma za zadanie przygotować środowisko, w którym inne programy mogą wykonywać pożyteczne funkcje.
- System operacyjny jako **rozszerzona maszyna**
    - architektura komputerów na poziomie języka maszynowego jest trudna do użycia w programach (szczególnie: operacje wejścia/wyjścia)
    - zadaniem systemu operacyjnego jest ukrycie tej złożoności i dostarczenie programiście bardziej przyjaznego interfejsu
    - system operacyjny udostępnia maszynę rozszerzoną / maszynę wirtualną, łatwiejszą do programowania
- System operacyjny jako **zarządca zasobów**
    - różne rodzaje zasobów w systemie: procesory, pamięć, zegary, dyski, terminale, napędy, taśmy magnetyczne, drukarki itd.
    - komputer ma za zadanie udostępniać zasoby użytkownikowi, kontrolować dostęp do nich i zapobiegać chaosowi i konfliktom między programami (i użytkownikami)
- System operacyjny jako **program sterujący**:
    - Nadzoruje działanie programów użytkownika, przeciwdziała błędom i zapobiega niewłaściwemu użyciu komputera.

Cele systemu operacyjnego:

- Wygoda użytkowania
- Wydajność efektywne wykorzystanie systemu komputerowego (szczególnie istotne w rozbudowanych, wielodostępnych systemach z podziałem czasu)

Oczywiście, często te cele są ze sobą sprzeczne. Początkowo stawiano głównie na wydajność, obecnie głównie przedkłada się wygodę użytkowania nad efektywnością.

Charakterystyka systemów wsadowych

Cecha charakterystyczna: Brak bezpośredniego nadzoru ze strony użytkownika podczas wykonywania jego zadania

Struktura systemu komputerowego:

- wielkie fizycznie maszyny obsługiwane przez konsolę
- urządzenia wejściowe: czytniki kart i przewijaki taśm
- urządzenia wyjściowe: drukarki wierszowe, przewijaki taśm, perforatory kart
- użytkownik przygotowywał zadanie (program, dane i informacje sterujące zapisane na kartach) i przekazywał je operatorowi
- wyniki były uzyskiwane po pewnym czasie (min., godz., dni)

Struktura systemu operacyjnego:

- SO rezyduje na stałe w pamięci operacyjnej
- obowiązkiem SO jest automatyczne przekazywanie sterowania od jednego zadania do następnego
- w celu przyspieszenia przetwarzania, zadania o podobnych wymaganiach grupowane razem i wykonywane w formie tzw. wsadu (batch), sortowanie dokonywane przez operatora
- jednostka centralna często pozostaje bezczynna w wyniku wolnej pracy mechanicznych urządzeń WE/WY

Modyfikacje:

- wprowadzono dyski , co pozwoliło na zastosowanie spooling'u (Simultaneous Peripheral Operation On-Line – jednoczesna, bezpośrednia praca urządzeń)
- spooling pozwala na jednoczesne wykonywanie obliczeń jednego zadania i operacji WE/WY innego
- dysk – bufor, pozwala na czytanie z maksymalnym wyprzedzeniem z urządzeń WE i przechowywanie plików wyjściowych, aż urządzenia WY będą je mogły przyjąć
- zawartość karty nie była po odczytaniu z czytnika ładowana bezpośrednio do pamięci, ale przechowywana na dysku, a system operacyjny przechowywał tablicę opisującą rozmieszczenie obrazów kart na dysku
- podobnie – komunikaty wyjściowe były zapisywane do bufora systemowego i na dysku, a drukowane dopiero po zakończeniu zadania

Wady systemów wsadowych:

- użytkownik nie może ingerować w zadanie podczas jego wykonywania się, musi zatem przygotować karty sterujące dla wszystkich możliwych zdarzeń.
- następne kroki wykonania zadania mogą zależeć od wcześniejszych wyników (np. kompilacja, uruchamianie...)
- trudności w testowaniu, programista nie może na bieżąco zmieniać programu w celu obserwacji jego zachowań

Charakterystyka SO z podziałem czasu i systemów interakcyjnych

System z podziałem czasu charakteryzuje się tym, że jest w nim wielu użytkowników, każdy uzyskuje dostęp do procesora przez pewną małą porcję czasu. Każdy użytkownik ma przynajmniej jeden proces w pamięci.

System interakcyjny umożliwia bezpośredni dialog użytkownika z systemem oraz bezpośredni dostęp do plików. Zadanie interakcyjne składa się z wielu krótkich działań, a rezultaty poszczególnych poleceń mogą być nieprzewidywalne(w przeciwieństwie do na przykład wsadowych).

SO czasu rzeczywistego i ich rodzaje.

System operacyjny czasu rzeczywistego (ang. Real Time Operating System RTOS):

- Stosowane, gdzie istnieją surowe wymagania na czas wykonania operacji lub przepływu danych
- Przykład: sterownik w urządzeniu np. przy nadzorowaniu eksperymentów naukowych, obrazowaniu badań medycznych, sterowaniu procesami przemysłowymi , systemach wizualizacji komputer pozyskuje dane z czujników i musi je analizować i regulować działanie kontrolowanego obiektu, w zależności od jego stanu
- **Wymaganie systemu czasu rzeczywistego**: przetwarzanie danych **musi** się zakończyć przed upływem określonego czasu

Rodzaje systemów czasu rzeczywistego:

- **rygorystyczny system czasu rzeczywistego** (ang. hard real-time system)

    - gwarantuje terminowe wypełnianie krytycznych zadań, systemy o specjalnej konstrukcji np. dane w pamięci o krótkim czasie dostępu lub w pamięci ROM, z reguły brak pamięci wirtualnej.
    - (takie, dla których znany jest najgorszy (najdłuższy) czas odpowiedzi, oraz wiadomo jest, że nie zostanie on przekroczony.)
    - Żaden z istniejących, uniwersalnych systemów operacyjnych nie umożliwia działania w czasie rzeczywistym.
- **łagodny system czasu rzeczywistego** (ang. soft real time system)
    - krytyczne zadanie do obsługi w czasie rzeczywistym otrzymuje pierwszeństwo przed innymi zadaniami i zachowuje je aż do swojego zakończenia, opóźnienia muszą być ograniczone --- zadanie nie może czekać w nieskończoność na usługi jądra.
    - (takie, które starają się odpowiedzieć najszybciej jak to możliwe, ale nie wiadomo jest, jaki może być najgorszy czas odpowiedzi.)
    - Zastosowanie w przemyśle i robotyce jest ryzykowne, przydatne w technikach multimedialnych, kreowaniu sztucznej rzeczywistości, zaawansowanych projektach badawczych (wyprawy planetarne, badania podmorskie). Większość współczesnych systemów operacyjnych (w tym Unix) może spełniać te wymagania

Budowa SO z jądrem warstwowym

Stosowane w celu ułatwienia modyfikacji systemu poprzez jego modularyzację.

Sposób modularyzacji:
- dzielenie systemu operacyjnego na warstwy (poziomy)
- każda nowa warstwa jest zbudowana powyżej warstw niższych
- najniższa warstwa (0)– sprzęt
- najwyższa warstwa (N)– interfejs użytkownika
- dowolna warstwa pośrednia M:
    - zawiera dane i procedury, które mogą być wywołane z warstw wyższych (M+1)
    - może wywoływać operacje dotyczące warstw niższych
- podejście warstwowe ułatwia wyszukiwanie błędów i weryfikację systemu

Możliwe wady realizacji warstwowych: mniejsza wydajność, obecnie ogranicza się liczbę warstw.

Przykład systemu warstwowego: system THE
- Warstwa 5: Programy użytkowe
- Warstwa 4: Buforowanie urządzeń wejścia i wyjścia
- Warstwa 3: Program obsługi kontroli operatora
- Warstwa 2: Zarządzanie pamięcią
- Warstwa 1: Planowanie przydziału procesora
- Warstwa 0: Sprzęt

Mikrojądro - wady i zalety

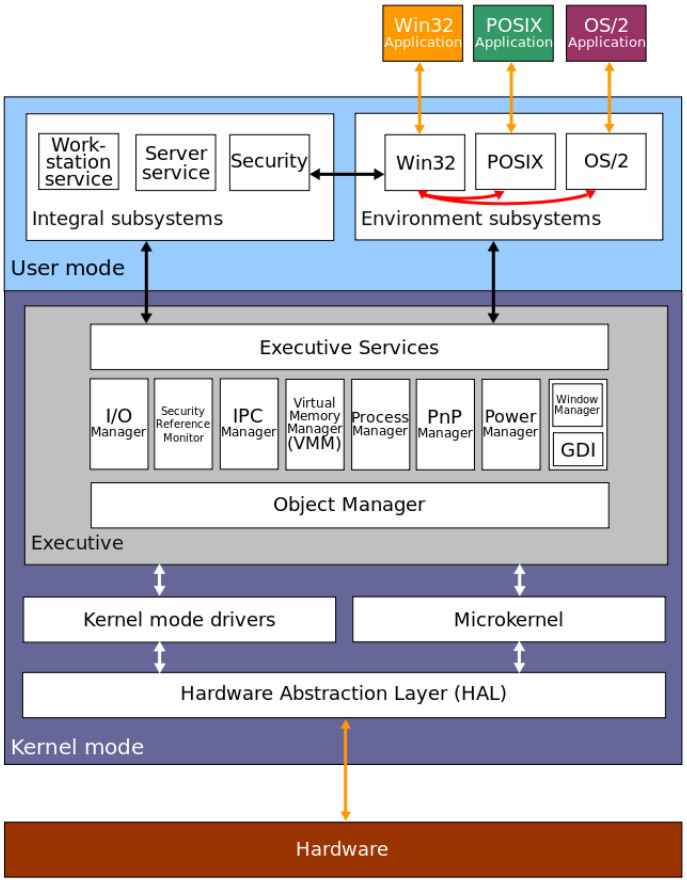
Zalety:
● **Jednolite interfejsy-** procesy nie muszą odróżniać usług jądra od usług użytkownika, gdyż wszystkie usługi są udostępniane za pomocą przekazywania komunikatów.
● **Rozszerzalność** – architektura mikrojądra sprzyja rozszerzalności, gdyż pozwala dodawać nowe usługi oraz obsługiwać zróżnicowane istniejące usługi; dodanie nowej funkcji wymaga dodania nowego serwera lub modyfikacji istniejącego, nie trzeba generować nowego jądra

● **Elastyczność** – można zarówno dodawać nowe usługi i rozbudowywać system jak też usuwać pewne usługi, aby uzyskać mniejszą i wydajniejszą realizację
● **Przenośność między platformami-** całość lub duża część kodu zależnego od architektury sprzętowej znajduje się w mikrojądrze, łatwość przenoszenia na inną architekturę, gdyż trzeba zmieniać tylko dobrze określone, nieliczne moduły
● **Stabilność-** niewielkie mikrojądro może być dokładnie przetestowane
● **Obsługa systemów rozproszonych-** komunikaty skierowane do serwera usługi mogą dotyczyć zarówno serwerów zlokalizowanych na tym samym komputerze jak i na innych
● **Obsługa systemów obiektowych** – rozwiązania obiektowe wymuszają większą dyscyplinę podczas projektowania mikrojądra oraz modularnych rozszerzeń.

Wady:
● **Komunikacja-** intensywnie wykorzystywane jest jądro. Komunikacja klient-serwer np. manager okienek > system plików (system plików jest tutaj usługą serwerem) wygląda tak: klient (wysyła zapytanie) > jądro > serwer (wysyła odp) > jądro > klient. Komunikacja między jądrem a serwerami użytkownika wymaga przełączania kontekstów, co jest bardzo kosztowne.
● **Synchronizacja** w przypadku synchronizacji działań dokonywanych przez serwery poziomu użytkownika f

Omów budowę systemu Windows NT

Będziemy się głównie opierać na starszych wersjach, tzn. 2000, XP; diagram jest z Windowsa 2000. Z tego co wiem to sporo rzeczy jest od Visty konsekwentnie przenoszonych z kernel-space do user-space (GDI i renderowanie czcionek [serio, to jest przyczyna dlaczego tak łatwo można było atakować Windowsa czcionkami :D], Window Manager -> DWM, User Mode Drivers). Ten podsystem POSIX to całkowicie co innego niż podsystem Linux z Windowsa 10 - tak, to już kiedyś było i nawet działało lepiej. Sporo rzeczy napisałem tutaj z głowy, mogłem się pomylić ~Marek



Jądro Windows NT jest tzw. **hybrydowe** (kontrowersyjne określenie), tzn. jest to proste jądro **monolityczne** (na diagramie *microkernel*), z resztą komponentów umiejscowioną w obrębie **modułu wykonawczego** (*executive*), działającego w kernel-space (czyli podobnie do usług w mikrojądrze, ale tamte są w user-space).

Po kolei wszystkie warstwy:

1. HAL (ang. Hardware Abstraction Layer) - warstwa abstrakcji chowająca zawiłości różnych architektur sprzętowych pod ujednoliconym API. Mogą z niej korzystać zarówno jądro jak i sterowniki. Ta warstwa nie jest ściśle rozgraniczona z jądrem, występuje komunikacja obustronna.

   Rzecz przez którą odrzucono wbudowane otwartoźródłowe sterowniki kart graficznych AMD w Linuxie :) (bo Linux czegoś takiego nie ma, a AMD wzięło sterowniki z Windowsa i wprowadziło takie własne HAL)
2. **Jądro i Kernel-mode drivers** - znajdują się równolegle pomiędzy HAL a Executive

   Jądro - odpowiada za szeregowanie zadań, harmonogram realizacji wątków, obsługę sytuacji wyjątkowych i synchronizację pracy wieloprocesorowej; nic więcej. Jądro samo z siebie nie jest stronicowane ani wywłaszczalne (w przeciwieństwie do reszty modułów). Sam też jest odpowiedzialny za synchronizację swoich sekcji krytycznych.

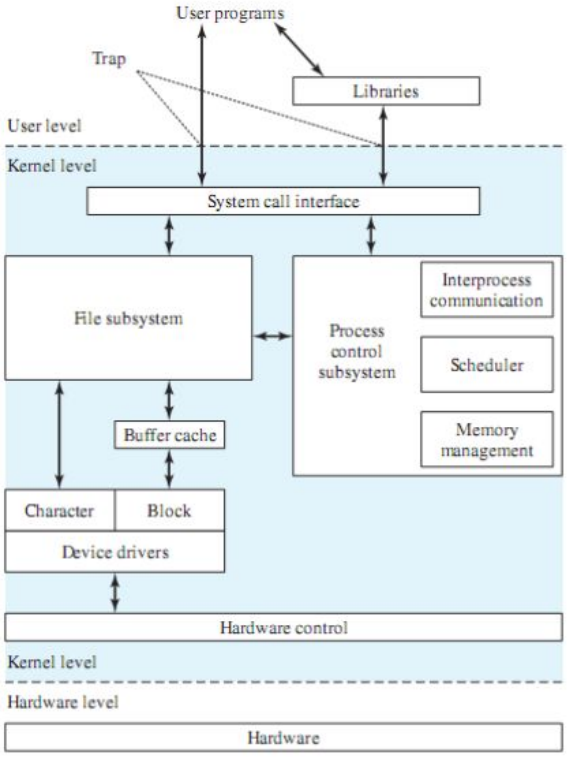
   Kernel-mode drivers - niskopoziomowe sterowniki działające w trybie jądra, można też robić tzw. **user-mode drivers** które działają w trybie użytkownika, są prostsze i bezpieczniejsze ale też wolniejsze
3. **Executive**, moduł wykonawczy - spory zbiór usług, działających w trybie jądra, udostępniających prawie wszystkie elementarne (i nie tylko) funkcjonalności systemu.

   Na Executive składa się wiele komponentów, podstawowe to:
   a. **Object Manager** - tak jak w Unixie wszystkie zasoby reprezentuje plik, tak w Windowsie reprezentują **obiekty**. Jest to bardzo zbliżony model do tego co znamy z programowania obiektowego.
   b. **Cache Controller** - wspólne miejsce do obsługi pamięci podręcznej pamięci, I/O i sterowników
   c. **Configuration Manager** - implementuje Rejestr (regedit i te sprawy)
   d. **I/O Manager** - implementuje niezależnie od sprzętu całą obsługę wejścia/wyjścia i dysponuje sterownikami urządzeń
   e. **Local Procedure Call (LPC)** - taki mechanizm IPC
   f. **Memory Manager** - zarządza pamięcią wirtualną, ochroną pamięci, stronicowaniem
   g. **Process Structure** - obsługuje tworzenie i usuwanie procesów i wątków (wykonywanie to zadanie jądra). Implementuje koncept *Job*'a, grupy procesów które mogą być usunięte jako całość, coś jak linkowanie w Erlangu.
   h. **PnP Manager (Plug and Play)** - obsługuje wykrywanie urządzeń i automatyczne wczytywanie sterowników. Sporo funkcjonalności działa też w obszarze użytkownika, gdzie np. odpalany jest Windows Update w celu pobrania brakujących sterowników
   i. **Power Manager** - obsługuje zdarzenia związane z zasilaniem, usypia CPU, kontroluje sterowniki za pomocą specjalnych przerwań
   j. **Security Reference Monitor** - obsługuje wszystkie sprawy związane z egzekwowaniem uprawnień itp.
   k. **GDI** - mój ulubieniec. Prastary framework do rysowania, GUI i renderowania czcionek. Siedzi w kernelu bo dawno temu dawało to sporą wydajność.

4. **-- User-space --**
   Podsystemy środowiskowe - jedną z ciekawszych cech Windows NT jest natywna zdolność do udawania przeróżnych środowisk. Dominujący z nich to słynne **Win32**. Dawniej (<= Windows 2000) funkcjonowały też podsystemy **OS/2** i **POSIX** (można się domyślić co robiły). W Windows 10 pojawił się **Windows Subsystem for Linux**.

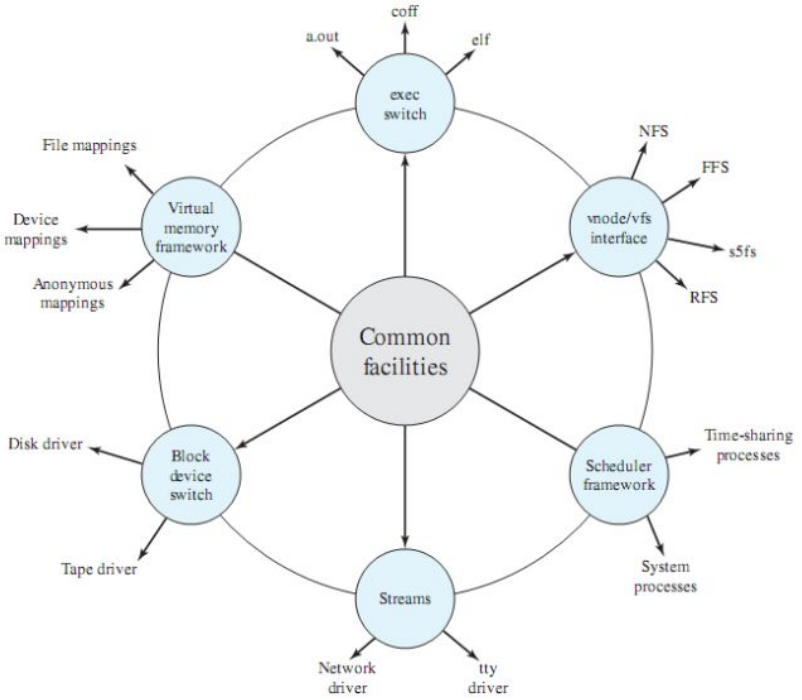
Omów budowę tradycyjnego jądra UNIX oraz budowę Systemu V Release 4

Unix:



-Sprzęt połączony z kontrolerem sprzętu
-Drivery do urządzeń zewnętrznych
-Character/Block (rodzaje urządzeń jedne mogą być odczytywane po charze np. klawiatura, inne blokami np. dyski,taśmy do obu potrzebujemy driverów)**ktoś coś???????????**
-Buffer cache - zachowuje ostatnio pobrane dane, dzięki czemu przyspiesza wolne operacje pobieranie danych z dysku.
-Podsystem plików - zajmuje się dostępem do pamięci
- Sygnały/Pułapki - sygnały pozwalają na komunikację asynchroniczną, a pułapki na sprzątanie przed zakończeniem programu przez sygnał

SVR4:
W porównaniu do tradycyjnego jądra UNIX, w VR4 wprowadzono następujące usprawnienia:
●możliwość wykonywania różnego rodzaju plików wykonywalnych
●osobne sposoby obsługi procesów z podziałem czasu i procesów realtime
●wprowadzenie vnodów, co pozwala na obsługę wielu systemów plików



“W latach siedemdziesiątych i wczesnych osiemdziesiątych jądra systemów uniksowych nie były zbyt wszechstronne. Wspierały one tylko jeden system plików, jeden algorytm szeregowania i jeden format pliku wykonywalnego. Jedynym konfigurowalnym elementem były tablice rozdzielcze (ang. switch) urządzeń znakowych i blokowych, które umożliwiały na dostęp do różnych typów urządzeń za pomocą wspólnego interfejsu. (…)

Coraz powszechniejsze stosowanie systemów uniksowych spowodowało potrzebę zaprojektowania bardziej elastycznego systemu, który mógłby wykonywać to samo zadanie na wiele różnych sposobów. Zaowocowało to opracowaniem wielu elastycznych struktur, takich jak interfejs v-węzłów, tablica rozdzielcza dla funkcji exec, klasy szeregowania i segmentowa architektura pamięci. Nowoczesne jądro uniksowe (czyt. sv4) przypomina system z rys. powyżej. Każdy z zewnętrznych okręgów przedstawia interfejs, który można zaimplementować na wiele różnych sposobów.” ~ Uresh Vahalia “Jądro systemu UNIX” (str. 15-16)

Wykład 2 - Zarządzanie procesami

Podział czasu dla wątków w systemie wieloprocesorowym

Różnica w porównaniu z planowaniem procesów:

- **dla realizacji jednoprocesorowych:** wątki ułatwiają strukturalizację programu oraz umożliwiają wykonywanie procesu nawet wtedy, gdy jakiś jego wątek czeka na operację we/wy
- **dla realizacji wieloprocesorowych:** wątki wchodzące w skład jednego procesu zwykle blisko współpracują ze sobą, a zatem ich równoległe wykonywanie może znacząco zwiększać szybkość
    - Wątki nie muszą być blokowane w oczekiwaniu na wykonanie akcji przez inne, co powoduje wykonanie kosztowych funkcji systemowych oraz zmianę kontekstu wykonania na procesorach
    - Dla aplikacji wymagających znacznych interakcji między wątkami, niewielkie zmiany w zarządzaniu i szeregowaniu wątków mogą znacząco zmienić wydajność aplikacji

Podstawowe podejścia:

1. **Współdzielenie obciążenia (Load Sharing)** – stosowana jest globalna kolejka gotowych do wykonania wątków, procesor gdy jest wolny wybiera wątek z kolejki
2. **Szeregowanie grupowe (zespołowe) (Gang scheduling)** – zbiór wątków tego samego procesu jest jednocześnie wykonywany na wielu procesorach
3. **Rezerwacja procesorów (Dedicated processor assignment)** – przydział wątków do wykonania na konkretnych procesorach
4. **Dynamiczne szeregowanie (Dynamic scheduling)** – ilość wątków w procesach może być zmieniana w trakcie wykonania

Rodzaje wątków i ich specyfikacja

1. **Wątki poziomu użytkownika** - czyli to co widzi aplikacja
    - abstrakcja wątków jest udostępniana na poziomie użytkownika, jądro nie wie o ich istnieniu
    - realizacja za pomocą pakietów bibliotecznych jak pthreads (zgodny ze standardem POSIX)
    - synchronizacja, szeregowanie i zarządzanie takimi wątkami odbywa się bez udziału jądra – jest bardzo szybkie

- kontekst wątku z poziomu użytkownika jest zapamiętywany i odtwarzany bez udziału jądra
- wątek użytkownika ma własny stos, przestrzeń do zapisania kontekstu rejestrów z poziomu użytkownika a także informacje o stanie wątku jak maska sygnałów

Szeregowanie:

- jądro szereguje procesy lub procesy lekkie, w obrębie których wykonują się wątki użytkownika
- proces do szeregowania swoich wątków stosuje funkcje biblioteczne
- jeśli jest wywłaszczany proces, lub proces lekki, są wywłaszczane także jego wątki
- jeśli watek użytkownika wykona blokującą funkcję systemowa, jest wstrzymywany proces lekki, w którym wykonuje się wątek - jeśli w obrębie procesu jest tylko jeden proces lekki, to wszystkie jego wątki są wstrzymane

Wątki jądra:

- wątek jądra nie musi być związany z procesem użytkownika
- jądro tworzy go i usuwa wewnętrznie w miarę potrzeb
- wątek odpowiada za wykonanie określonej czynności, współdzieli tekst jądra i jego dane globalne, jest niezależnie szeregowany i wykorzystuje standardowe mechanizmy jądra, takie jak *sleep()* czy *wakeup()*
- wątki potrzebują jedynie następujących zasobów:
  - własnego stosu,
  - przestrzeni do przechowywania kontekstu rejestrów w czasie, gdy wątek nie jest wykonywany
- tworzenie i stosowanie wątków jądra nie jest kosztowne
  - przełączanie kontekstu między wątkami jądra jest szybkie, ponieważ nie trzeba zmieniać odwzorowań pamięci
- zastosowania wątków jądra
  - wątki jądra są przydatne do wykonywania pewnych operacji takich, jak asynchroniczne wejście-wyjście, zamiast udostępniać osobne operacje asynchronicznego we/wy, jądro może realizować je, tworząc odrębny wątek do wykonania każdego zlecenia
  - wątki poziomu jądra można wykorzystywać też do obsługi przerwań

2. **Procesy lekkie** - mechanizm pozwalający na wieloprocesorowe wykonywanie wątków poziomu użytkownika
   - wysokopoziomowe pojęcie abstrakcyjne oparte na wątkach jądra - system udostępniający procesy lekkie, musi także udostępniać wątki jądra
   - każdy proces lekki jest związany z wątkiem jądra, ale niektóre wątki jądra są przeznaczone do realizacji pewnych zadań systemowych, wtedy nie przypisuje się im żadnych lekkich procesów
   - w każdym procesie może być kilka procesów lekkich, każdy wspierany przez oddzielny wątek jądra, współdzielą one przestrzeń adresową i inne zasoby procesu
   - procesy lekkie są niezależnie szeregowane przez systemowego planistę
   - procesy lekkie mogą wywoływać funkcje systemowe, które powodują wstrzymanie w oczekiwaniu na we/wy lub zasób
   - proces działający w systemie wieloprocesorowym może uzyskać rzeczywistą równoległość wykonania, każdy proces lekki może być uruchamiany na oddzielnym procesorze
   - wielowątkowe procesy są przydatne wtedy, gdy każdy wątek jest w miarę niezależny i rzadko porozumiewa się z innymi wątkami

Omów modele wielowątkowości

1. **1:N** - scheduler jądra (KS) widzi 1 proces, natomiast scheduler wątków użytkownika (UTS) procesu (każdy ma swój) widzi *N* wątków. Komunikacja KS-UTS nie jest wymagana.
2. **1:1** - wszystkie procesy i wątki dla KS to to samo, UTS jest nie potrzebny. Konieczna jest organizacja tworzenia, synchronizacji i pamięci dzielonej za pomocą bibliotek.
3. **M:N** - model *M* procesów lekkich widzianych przez KS i *N* wątków widzianych przez UTS. *M*<*N*, jeżeli równe to mamy do czynienia z 1:1. Najtrudniejsza implementacyjnie ale też najbardziej wszechstronna, wymaga komunikacji KS-UTS.

Podaj sposoby szeregowania (systemy jednoprocesorowe)

[dopytywał się głównie o wielokolejkowe]
jest w opracowaniu, jest na slajdach, nie chce mi się przepisywać bo dużo

Proces a program. Budowa procesu i budowa wątku.

[co do procesów to trzeba znać struct task i jeszcze inne rzeczy]

Proces a program:

Program	Proces
- jakiś zapisany kawałek bajtów który da się wykonać, np. plik wykonywalny na dysku (obraz początkowego stanu procesu) - obiekt pasywny	- wykonujący się program - obiekt aktywny - zawiera licznik rozkazów określający następny rozkaz do wykonania i zbiór przydzielonych zasobów

Budowa procesu:
Proces to wykonujący się program który składa się z:
- **kodu programu** (sekcji tekstu, ang. text section)
- odpowiedniego ustawienia **licznika rozkazów** (program counter)
- **zawartości rejestrów** procesora
- **stosu procesu** (ang. process stack) z danymi tymczasowymi (parametrami procedur, adresami powrotnymi, zmiennymi tymczasowymi)
- **sekcji danych** (ang. data section) ze zmiennymi globalnymi

Każdy proces jest reprezentowany w systemie operacyjnym przez **blok kontrolny procesu** (ang. process control block - PCB, zamiennie nazywany **blok kontrolny zadania**). W skład PCB wchodzą:
- **stan procesu**
- **licznik rozkazów**
- **rejestry procesora**
- **informacje o planowaniu przydziału procesora** - m.in. priorytet, wskaźniki do kolejek
- **informacje o zarządzaniu pamięcią** - m. in. tablice stron
- **informacje do rozliczeń** - m. in. ograniczenia czasowe, statystyki zużycia czasu procesora
- **informacje o stanie I/O** - m. in. wykaz deskryptorów

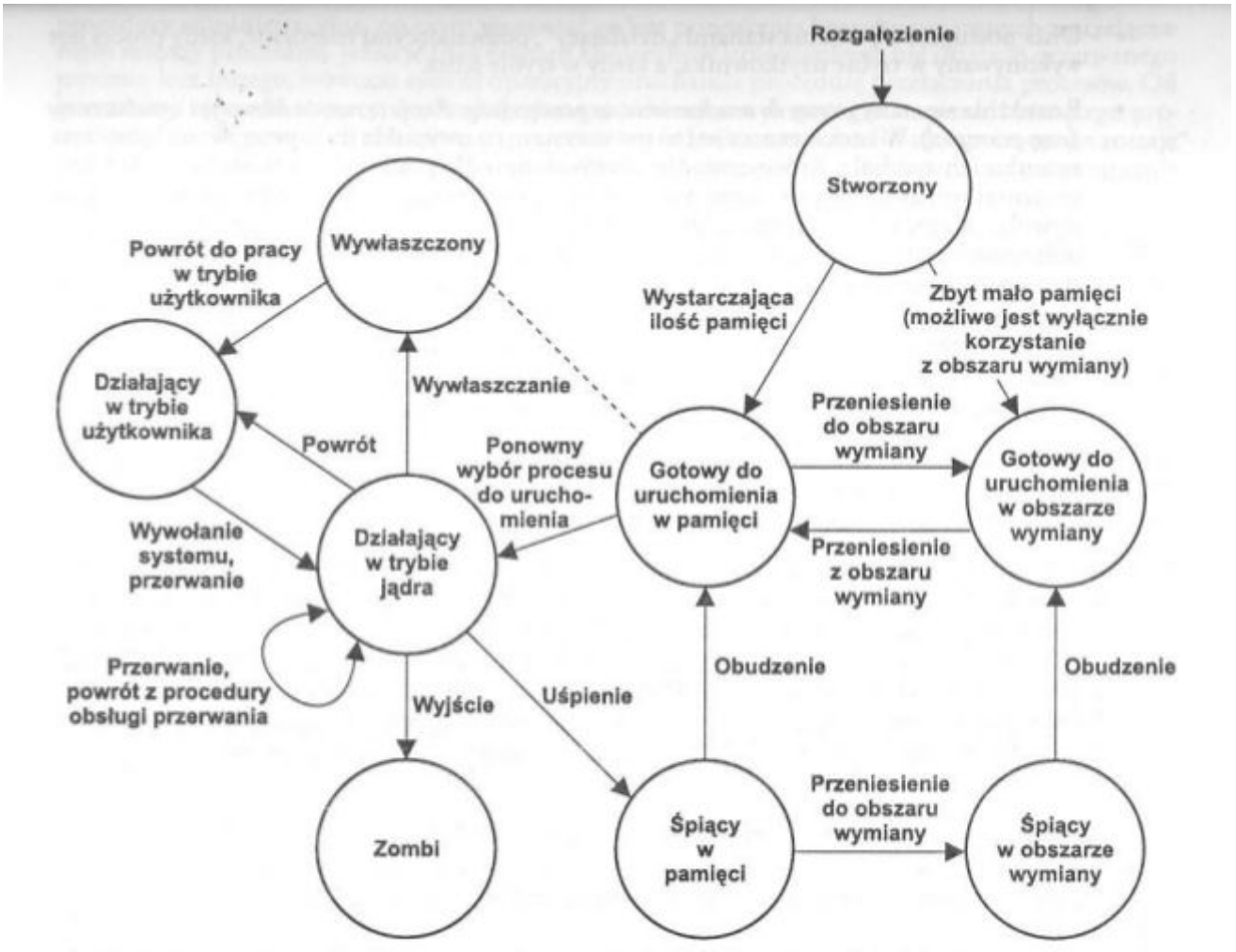
W Linuxie PCB to struktura **`task_struct`**, która zawiera także m. in.:
- **maskę sygnałów**

- **przypisany terminal** (TTY)

Budowa wątku:
Wątek to podstawowa jednostka wykorzystania procesora, która obejmuje:
- licznik rozkazów
- zbiór rejestrów
- stos

wiele wątków może współdzielić:
- sekcję kodu
- sekcję danych
- zasoby systemu

Stany procesów. Przejścia między stanami.



Wykład 3 - Zarządzanie procesami i synchronizacja

Rozwiązanie problemu 5 filozofów

Rozwiązanie problemu, gdy ograniczmy liczbę filozofów przebywających w jadalni:

```
Jadalnia: Semafor:=4;
Widelec: array(0..4) of Semafor;
task body Filozof is
begin
    loop
        Myśl;
        Czekaj(Jadalnia);
        Czekaj(Widelec(I));
        Czekaj(Widelec((I+1) mod 5));
        Jedz;
        Sygnalizuj(Widelec(I));
        Sygnalizuj(Widelec((I+1) mod 5));
        Sygnalizuj(Jadalnia);
    end loop;
end Filozof;
```

Co to jest sekcja krytyczna? Po co jest?wewnętrznej fragmentacji

[Polska Wiki traktuje tutaj o wątkach, a angielska i Koźlak o procesach]

Sekcja krytyczna – fragment kodu programu, w którym korzysta się z zasobu dzielonego. Tylko jeden proces jednocześnie może przebywać w swojej sekcji krytycznej.

Sekcje krytyczne chronią przed równoległym dostępem zasoby nieprzystosowane do tego. Przystępnie obrazuje to diagram po prawej.

Warunki jakie musi spełniać sekcja krytyczna

[Technicznie: Warunki jakie musi spełniać rozwiązanie problemu sekcji krytycznej]

1. **Wzajemne wykluczanie** - jeżeli proces P_i działa w swojej sekcji krytycznej, to żaden inny proces nie działa w sekcji krytycznej
2. **Postęp** - jeżeli żaden proces nie działa w sekcji krytycznej i istnieją procesy, które chcą wejść do sekcji krytycznych, to tylko procesy niewykonujące swoich reszt (tj. sekcji niekrytycznych) mogą kandydować jako następne do wejścia do sekcji krytycznych i wybór ten nie może być odwlekany w nieskończoność.
3. **Ograniczone czekanie** - musi istnieć wartość graniczna liczby wejść innych procesów do ich sekcji krytycznych po tym, gdy dany proces zgłosił chęć wejścia do swojej sekcji krytycznej i zanim uzyskał na to pozwolenie.

   Chodzi o to że proces nie może czekać w nieskończoność na wejście w sekcję krytyczną.

Poprawny algorytm rozwiązania problemu sekcji krytycznej dla dwóch procesów

To coś (prawdopodobnie oczekiwane) albo algorytm piekarni albo semafory:

```
{ Procesy: P0 i P1 }
j = 1-i

Proces i:

{określa, że dany proces jest gotowy do wejścia sekcji krytycznej}
var znacznik: array[0..1] of boolean;

{określa, który proces ma prawo do wejścia do sekcji krytycznej}
var numer: 0..1;

znacznik[0] := false;
znacznik[1] := false;

numer := dowolna;

repeat
    znacznik[i] := true;
    numer := j;

    { czekaj na możliwość wejścia w sekcję krytyczną }
    while (znacznik[j] and numer=j) do nic;

    {- sekcja krytyczna -}

    znacznik[i] := false;

    {- reszta -}
until false;
```

Algorytm piekarni

Zasady:

1. przy wejściu do sklepu klient dostaje numerek
2. obsługę rozpoczyna się od klienta z najmniejszym numerem
3. algorytm nie gwarantuje, że dwa procesy (klienci) nie dostaną tego samego numeru
4. w przypadku kolizji pierwszy zostanie obsłużony proces o wcześniejszej nazwie

Algorytm (z Wikipedii, ten ze slajdów jest niezjadliwy):

```
// deklaracja i nadanie początkowych wartości zmiennych globalnych
Wpisywanie: array [1..N] of bool = {false};
Numer: array [1..N] of integer = {0};

blokuj(integer i) {
    Wpisywanie[i] = true;
    Numer[i] = 1 + max(Numer[1], ..., Numer[N]);
    Wpisywanie[i] = false;
    for (j = 1; j <= N; j++) {
        // Czekaj, aż proces j dostanie swój numer:
        while (Wpisywanie[j])
        { /* Nie rób nic. */ }
        // Czekaj na wszystkie procesy z numerami mniejszymi, bądź równymi,
        // ale z wyższym priorytetem, aż się zakończą:
        while ((Numer[j] != 0) && ((Numer[j], j) < (Numer[i], i)))
        { /* Nie rób nic. */ }
    }
}

odblokuj(integer i) {
    Numer[i] = 0;
}

Proces(integer i) {
    while (true) {
        blokuj(i);
        // Miejsce na sekcję krytyczną...
        odblokuj(i);
        // Miejsce na sekcję lokalną...
    }
}
```

Algorytm czytelników i pisarzy - implementacja na monitorach

```
monitor Monitor_czytelnicy_pisarze is
    Czytelnicy: Integer:=0; {Licznik czytelników, którzy czytają}
    Pisanie: Boolean:=False; {Czy jakiś proces pisze}
    Można_czytać, można_pisać: Warunek; {wstrzym. czyt. i pisarzy}
    procedure Zacznij_czytanie is
    begin
        if Pisanie or Niepusta(Można_pisać) then
            Czekaj(Można_czytać);
        end if
        Czytelnicy:=Czytelnicy+1;
        Sygnalizuj(Można_czytać);{Kaskada wznowień}
    end Zacznij_czytanie
    procedure Zakończ_czytanie is
    begin
        Czytelnicy:=Czytelnicy-1;
        if Czytelnicy=0 then
            Sygnalizuj(Można_pisać)
        end if
    end Zakończ_czytanie
    procedure Zacznij_pisanie is
    begin
        if Czytelnicy != 0 or Pisanie then
            Czekaj(Można_pisać);
        end if;
        Pisanie:=true;
    end Zacznij_pisanie
    procedure Koniec_pisania is
    begin
        Pisanie:=False;
        if Niepusta(Można_czytać) then
            Sygnalizuj(Można_czytać);
        else
            Sygnalizuj(Można_pisać)
        end if;
    end Koniec_pisania;
end Monitor_czytelnicy_pisarze
```

Monitory

[można tutaj zamiennie użyć procesów zamiast wątków imho]

Monitor - bezpieczna wątkowo klasa, obiekt bądź moduł, który wykorzystuje wzajemne wykluczenie w celu udostępnienia bezpiecznego dostępu do metody bądź zmiennej przez jeden lub więcej wątków. W dowolnym momencie czasowym z dowolnej metody monitora może korzystać tylko jeden wątek naraz. Zmienne monitora są prywatne.

Operacje kolejkowe monitora *(co to tak dokładnie do diabła jest!?)* - coś a'la zmienne warunkowe
Monitor może zawierać kolejkę wątków i udostępniać następujące metody:

- **wait() / delay()** - zawieszenie wywołującego metodę wątku, dodanie go do kolejki i opuszczenie monitora (odblokowanie go, ale dalej siedzimy zawieszenie w środku metody *wait*)
- **signal() / continue()** - zdjęcie pierwszego wątku z kolejki (jeżeli takowy jest) i natychmiastowe wznowienie go (tzn. zacznie pracować od instrukcji po wywołaniu metody *wait*) i przekazanie kontroli
- **clear()** - wyczyszczenie kolejki *[co się dzieje z zawieszonymi procesami? sądząc po zachowaniu broadcast to sobie takie zostają...]*
- **empty()** - sprawdza czy kolejka jest pusta

Są dwa sposoby realizacji tych operacji:

1. **monitory z sygnalizacją** (oryginalne podejście)
    - to co zostało opisane powyżej
    - ma wady:
        - gdy wątek wykonujący *signal* nie kończy swego działania z monitorem, wówczas potrzebne są dwa dodatkowe przełączenia stanu procesu – jego zawieszanie i wznawianie
        - szeregowanie wątków związanych z sygnałem musi być niezawodne
2. **monitory z powiadamianiem i rozgłaszaniem**
    - metoda *signal* zostaje zastąpiona metodą **notify()**, która nie przekazuje kontroli do wątku natychmiast (tzw. *wolny monitor*) [gdy będzie wolny a nie tzw chyba : P]
    - mamy nową metodą **broadcast()**, która zdejmuje i odblokowuje wszystkie wątki z kolejki

Zastosowanie semaforów.

1. rozwiązanie problemu sekcji krytycznej (mutex na wykonywanie)
2. sekwencyjne wykonywanie operacji w różnych procesach jedna po drugiej (P2 blokuje mutex przed swoją operacją, P1 zwalnia mutex po swojej operacji)

Semafory. Operacje na nich. Aktywne czekanie.

Semafor to jedno z narzędzi synchronizacji danych. Jest to specjalny typ zmiennej typu integer, pozwalający na następujące operacje:

1. **inicjalizacja** - nadanie jakiejś wartości początkowej
2. **P, wait** - atomowa dekrementacja semafora
3. **V, signal** - atomowa inkrementacja semafora

```
procedure V (S : Semaphore);
begin
  (* Operacja atomowa: inkrementacja semafora *)
  S := S + 1;
end;

  (* Operacja atomowa: dekrementacja semafora *)
procedure P (S : Semaphore);
begin
  (* Cała operacja jest atomowa *)
  repeat
    Wait();
  until S > 0;
  S := S - 1;
end;
```

Przedstawiona implementacja ma sporą wadę: oczekiwanie na zwolnienie semafora powoduje kręcenie się w kółko procesu oczekującego. Jest to tzw. **aktywne czekanie** (ang. busy waiting) - procesy usiłujące wejść do sekcji krytycznej muszą wykonywać instrukcje pętli marnując cykle procesora. Rozwiązanie: dodajemy do semafora kolejkę procesów oczekujących i budzimy je wtedy kiedy jest taka możliwość.

Wykład 4 - Zakleszczenia

Wymień i omów WK zakleszczeń

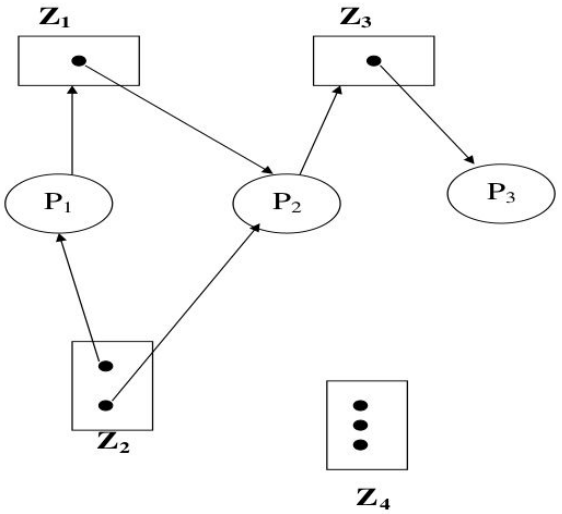
Do zakleszczeń może dochodzić, gdy zachodzą **jednocześnie** następujące warunki:

1. **Wzajemne wykluczanie** - przynajmniej jeden zasób musi być niepodzielny tzn., że zasobu tego może używać w danym czasie tylko jeden proces. Jeżeli inny proces zamawia dany zasób, to musi być opóźniony do czasu, aż zasób zostanie zwolniony.
2. **Przetrzymywanie i oczekiwanie** - musi istnieć proces, któremu przydzielono co najmniej jeden zasób i który oczekuje na przydział dodatkowego zasobu, przetrzymywanego właśnie przez inny proces
3. **Brak wywłaszczeń** - zasoby nie podlegają wywłaszczaniu, co oznacza, że zasób może być zwolniony tylko z inicjatywy przetrzymującego go procesu, po zakończeniu pracy tego procesu
4. **Czekanie cykliczne** - musi istnieć zbiór $\{P_0, P_1, ..., P_n\}$ czekających procesów, takich że P_0 czeka na zasób przetrzymywany przez P_1, P_1 czeka na zasób przetrzymywany przez proces P_2, … , P_{n-1} czeka na zasób przetrzymywany przez P_n, a P_n czeka na zasób przetrzymywany przez proces P_0. (warunek 4 implikuje 2)

Zdefiniuj graf przydziału zasobów. Jakie wnioski na temat zakleszczeń można z niego wyciągnąć?

Zakleszczenie można opisać za pomocą grafu skierowanego zwanego grafem przydziału zasobów systemu (ang. system resource-allocation graph):

- Zbiór wierzchołków jest podzielony na dwa rodzaje węzłów:
  - $P = \{P_1, P_2, ..., P_n\}$ - zbiór wszystkich **procesów systemu**, oznaczane ◯
  - $Z = \{Z_1, Z_2, ..., Z_m\}$ - zbiór wszystkich **typów zasobów systemowych**, oznaczane □, a każdy egzemplarz tego zasobu oznaczany kropką jest w prostokącie
- Są dwa rodzaje krawędzi:
  - **krawędź zamówienia** $P_i \rightarrow Z_j$ - proces P_i zamówił egzemplarz zasobu typu Z_j i obecnie czeka na ten zasób
  - **krawędź przydziału** $Z_j \rightarrow P_i$ - egzemplarz zasobu typu Z_j został przydzielony do procesu P_i



W oparciu o definicję grafu przydziału zasobów można wykazać, że:

- jeśli graf nie zawiera cykli, to w systemie nie ma zakleszczonych procesów
- jeśli graf zawiera cykl, to w systemie może dojść do zakleszczenia
- jeżeli zasób każdego typu ma tylko jeden egzemplarz, to zakleszczenie jest faktem

Na czym polega mechanizm zapobiegania zakleszczeniom

Gwarantuje się, że przynajmniej jeden z warunków koniecznych wystąpienia zakleszczenia nie jest spełniony. Możliwym skutkiem ubocznym jest słabe wykorzystanie urządzeń i ograniczona przepustowość systemu.

1. **Wzajemne wykluczenie**
    - obowiązuje tylko w przypadku niepodzielnych zasobów
    - zasoby dzielone nie powodują zakleszczeń
2. **Przetrzymywanie i oczekiwanie**
    - trzeba zagwarantować, że jeżeli proces zamawia zasób, to nie powinien on mieć żadnych innych zasobów
    - albo proces zamawia i dostaje wszystkie potrzebne zasoby zanim rozpocznie działanie
    - **wady:** głodzenie, niskie wykorzystanie zasobów
3. **Brak wywłaszczeń**
    - **Algorytm 1**
        - jeżeli proces chce jakiś zajęty zasób, to zabieramy mu wszystkie i usypiamy go
        - jeżeli wszystkie potrzebne procesowi zasoby są dostępne (zarówno stare jak i żądane), to przywracamy go
    - **Algorytm 2**
        - jeżeli proces chce jakiś zasób, który jest zajęty, to próbujemy “podebrać” go innemu procesowi, który czeka na inny zasób
        - proces może być wznowiony tylko wtedy, gdy otrzyma nowe zasoby i odzyska zasoby utracone podczas czekania
4. **Czekanie cykliczne**
    - zagwarantowanie, że czekanie cykliczne nigdy nie wystąpi przez wymuszenie całkowitego uporządkowania wszystkich typów zasobów i wymaganie, by każdy proces zamawiał je we wzrastającym porządku numeracji

Likwidacja zakleszczeń za pomocą modyfikacji grafu przydziału zasobów. Jaka modyfikacja grafu?

Co to jest stan bezpieczny i ciąg bezpieczny?

Kontekst : Unikanie zakleszczeń. Procesy muszą deklarować jakich zasobów będą używać.

Stan bezpieczny

- System jest w stanie bezpiecznym, gdy istnieje porządek, w którym może przydzielić (nawet maksymalną zadeklarowaną) liczbę zasobów każdemu procesowi w systemie.
- System w stanie bezpiecznym wtw gdy istnieje **ciąg bezpieczny**.
- Gdy system nie jest w stanie bezpiecznym, jest w **stanie zagrożenia**
- **Stan zagrożenia** może (nie musi) doprowadzić do **zakleszczenia**
- Gdy system jest w stanie bezpiecznym na pewno nie dojdzie do **zakleszczenia**

Ciąg bezpieczny:

Istnieje taki ciąg procesów w systemie $\{P_1,...,P_N\}$, że każdy proces może otrzymać zadeklarowane zasoby korzystając z zasobów wolnych, lub zasobów procesów poprzednich. Po zakończeniu procesu P_i proces P_{i+1} może użyć jego zasoby.

Wykład 5 - Zarządzanie pamięcią

Stronicowanie - tu z kolei o tablice stron i odwrotną tablicę stron

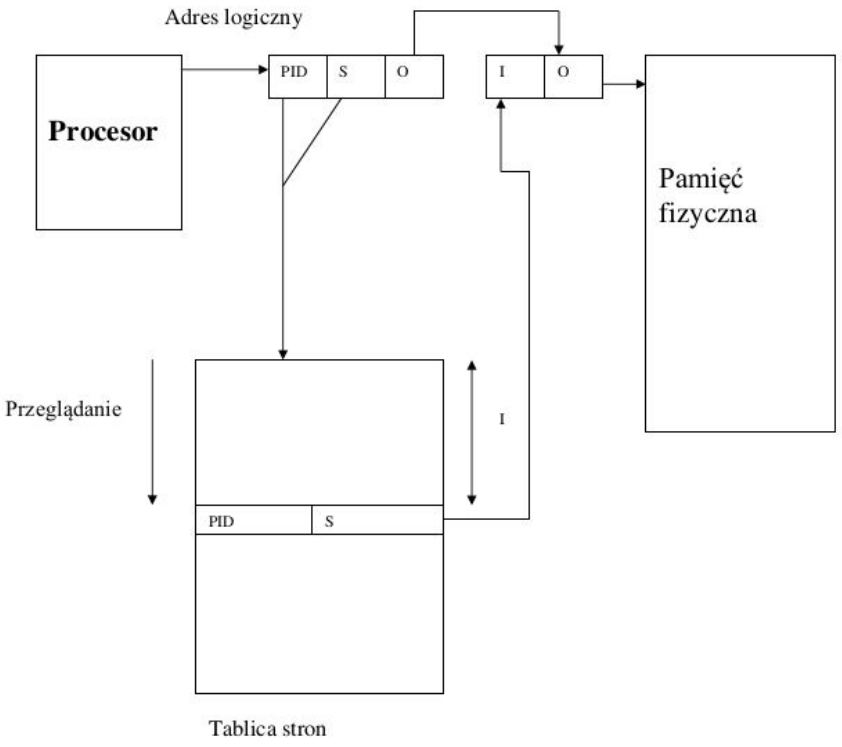
Stronicowanie to rozwiązanie, w którym proces widzi spójny obszar pamięci logicznej, ale nie tworzy ona spójnego obszaru w pamięci fizycznej. Zarówno pamięć logiczna, jak i pamięć fizyczna jest podzielona na kawałki **równej** wielkości. W odniesieniu do pamięci logicznej mówimy o stronach, a w odniesieniu do pamięci fizycznej mówimy o ramkach. Wielkość stron i ramek jest potęgą dwójki z przedziału od 512B do 16MB. Typowa wielkość to 4kB. Strony i ramki są podstawowymi jednostkami przydziału pamięci. W rezultacie, wielkość przydzielonego procesowi obszaru jest wielokrotnością wielkości ramek i stron. Strony pamięci logicznej mogą być umieszczone w dowolnych ramkach pamięci fizycznej. Nie muszą tworzyć spójnego obszaru w pamięci fizycznej, ani nie muszą występować w określonej kolejności.

Tablica stron zawiera adresy bazowe wszystkich stron w pamięci operacyjnej, jest używana do tłumaczenia adresu logicznego na adres fizyczny. Innymi słowy *tab[numer strony] = adres fizyczny tej strony*. **Każdy proces ma własną tablicę stron.** (dobrze mówię? inaczej nie widziałbym sensu odwrotnej tablicy tagggg)

Tablice stron mogą być wspomagane sprzętowo, np. za pomocą buforów translacji adresów stron (ang. translation look-aside buffers, TLB; używanych w Intelach x86), przechowujących ostatnio wyznaczane adresy fizyczne stron.

Odwrócona tablica stron jest tylko jedna. Zawiera tyle pozycji, ile ramek pamięci fizycznej jest faktycznie zainstalowanych w komputerze. Każda pozycja zawiera adres wirtualny strony przechowywanej w ramce rzeczywistej pamięci oraz informacje o procesie, do którego strona należy. Adres wirtualny stanowi trójka *(PID, numer strony, odległość)*, a wpis w odwróconej tablicy stron jest parą *(PID, numer strony)*.

Stosowana w x64, gdyż pełne zaadresowanie całej możliwej pamięci (16 etabajtów) wymagałoby tablicy o rozmiarze 32 petabajty dla stron o rozmiarze 4kb.



Algorytmy przydziału ciągłych fragmentów pamięci

1. **Pierwsze dopasowanie** (ang. first fit)
    - przydzielana jest pierwsza dziura o odpowiedniej wielkości
    - szukanie może być rozpoczęte od początku wykazu dziur, lub od punktu, w którym nastąpiło zatrzymanie
    - szukanie zostaje zakończone, gdy zostanie odnaleziona odpowiednia duża dziura
2. **Najlepsze dopasowanie** (ang. best fit)
    - przydziela się najmniejszą z dostatecznie dużych dziur
    - w poszukiwaniu odpowiedniej dziury, trzeba przejrzeć całą ich listę, o ile nie są one uporządkowane wg. rozmiarów
    - zapewnia najmniejsze pozostałości po przydziale
3. **Najgorsze dopasowanie** (ang. worse fit)§
    - przydziela się największą dziura
    - trzeba przeszukać całą listę dziur, o ile nie są one uporządkowane wg. wymiarów
    - pozostawia największą dziurę, która może w przyszłości być bardziej użyteczna niż mała pozostałość po najlepszym dopasowaniu

Zdefiniuj logiczną przestrzeń adresową oraz fizyczną przestrzeń adresową

- **adres logiczny** (ang. logical address) -- adres wytworzony przez procesor,
- **adres wirtualny** (ang. virtual address) -- inna nazwa adresu logicznego
- **adres fizyczny** (ang. physical address) -- rzeczywiste umiejscowienie w pamięci (adres umieszczony w rejestrze adresowym)
- **logiczna przestrzeń adresowa** (ang. logical address space) -- zbiór wszystkich adresów logicznych generowanych przez program
- **fizyczna przestrzeń adresowa** (ang. physical address space) -- zbiór wszystkich adresów fizycznych odpowiadających adresom logicznym z logicznej przestrzeni adresowej

Program użytkownika operuje na adresach logicznych, a nie fizycznych.

adres fizyczny = wartość bazy + adres logiczny

Sposoby wiązań

Wiązanie adresów - stowarzyszanie symbolicznych wyrażeń (zmiennych) występujących w programie z *adresami względnymi* (liczonymi od początku danego modułu) lub *bezwzględnymi*.

Dokonuje się na trzech etapach:

1. **podczas kompilacji** - *kod bezwzględny* - jeśli znane jest miejsce, gdzie proces będzie przebywał w pamięci, np. programu typu **.com* w DOS
2. **podczas ładowania programu do pamięci** - *kod przemieszczalny* - adresy są względne do pewnego miejsca, gdyż nie znamy położenia procesu w pamięci; można ładować kod pod różnymi adresami, zmieniając adres początkowy
3. **podczas wykonania** - proces może być przemieszczany w pamięci podczas wykonywania, konieczne jest czekanie z wiązaniem adresów aż do czasu wykonania

Sposoby wiązań/linkowania:

1. **ładowanie dynamiczne** - podprogram jest ładowany do pamięci dopiero gdy zostaje wywołany
    - nie ma konieczności wczytywania całego programu na raz
    - nie wymaga wsparcia ze strony systemu operacyjnego, jednak zazwyczaj dostarczane są funkcje biblioteczne
2. **konsolidacja statyczna** - biblioteki są na chama kopiowane do pliku wynikowego podczas kompilacji
3. **konsolidacja dynamiczna** - w pliku wynikowym umieszczane są tylko stuby linkowanych procedur, które wskazują jak odnaleźć odpowiedni podprogram biblioteczny lub jak go załadować, np. **.so*, **.dll*

Wykład 6 - Pamięć wirtualna

Co to jest szamotanie? Przyczyny, przeciwdziałanie

Szamotanie (ang. trashing) - sytuacja, w której proces musi dokonywać częstych wymian stron. Proces szamoce się, jeśli spędza więcej czasu na stronicowaniu, niż na wykonaniu.

Przykładowa sytuacja:

1. proces wchodzi w nową fazę działania i potrzebuje więcej ramek
2. wykazuje braki stron i powoduje utratę stron przez inne procesy
3. inne procesy także wykazują braki stron
4. wskutek oczekiwania procesów na przydział stron zmniejsza się wykorzystanie procesora
5. jeśli spada zużycie procesora, wtedy planista może chcieć zwiększyć poziom wieloprogramowości przyjmując nowe procesy i przydzielając im pamięć

Sposoby ograniczenia szamotania:

- lokalny (lub priorytetowy) algorytm zastępowania – szamoczący się proces nie doprowadza do szamotania innych procesów
- należy dostarczyć procesowi tyle ramek, ile potrzebuje
- model zbioru roboczego
    - ustalamy D - liczbę badanych ostatnich odwołań do ramek
    - dla każdego procesu tworzymy zbiór z ramek do których się odwoływał w ostatnich D odwołaniach - zbiór roboczy
    - następnie sumujemy moce (wielkości) zbiorów roboczych.
    - ograniczamy wieloprogramowość do takiego stopnia aby ta suma była mniejsza od liczby ramek w systemie
- mierzenie częstości braków stron (ang. page-fault frequency, PFF) - jeżeli proces ma wysoką ilość braków stron, to potrzebuje więcej ramek; jeżeli niską - to ma ich zbyt wiele

Algorytmy stosowane przy wymianie stron na żądanie

jest ich dużo, patrzać do opracowania z 2014 albo niech ktoś to opracuje ładnie bo mi się nie chce

- mi też nie