4. PROCESY i WĄTKI

Współbieżne wykonywanie programów (procesów) w **PAO wymaga pełnej kontroli nad nimi** i **odseparowania ich** od **siebie.**

Model systemu wykorzystujący **procesy** ułatwia to zadanie.

➔ Pojęcia **proces** i **wątek** używane są w odniesieniu do wykonywalnego kodu.

Proces ma przestrzeń adresową pamięci wirtualnej i informacje takie, jak priorytet, itp.

Proces ma jeden lub więcej **wątków**, które są zarządzane przez **jądro**.

Wątek ma własny stan, priorytet, przypisanie do procesora i informacje rozliczeniowe.

Proces: program w trakcie wykonywania.

Terminy: **zadanie** i **proces** są często używane zamiennie **!!!**

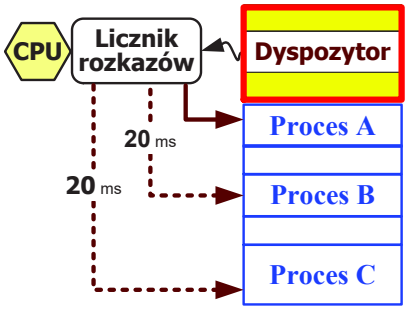
Program na dysku jest **zadaniem** czekającym na uruchomienie ~~a nie procesem~~.

System operacyjny zapewnia:

1. **przeplatanie** wykonywania procesów, celem maksymalizacji wykorzystania CPU;
2. **alokowanie** zasobów dla procesów zgodnie z przyjętymi zasadami;
3. **komunikację** między procesami i tworzenie procesów przez użytkownika.

Proces **wykonywany jest sekwencyjnie** -w dowolnej chwili na z●amówienie procesu może być wykonywany jeden rozkaz kodu programu.

Przetwarzanie poleceń wynika z kolejności narzucanej przez zmieniającą się wartość **licznika rozkazów**.



Ślad procesu *(trace)*: lista odpowiadającego mu ciągu rozkazów.

Zachowanie CPU charakteryzuje zestawienie przeplatających się **śladów procesów**.

Proces to nie tylko kod programu.

To wartość licznika **rozkazów** i **rejestrów** procesora,
to **stos procesu** przechowujący parametry funkcji,
to **adresy powrotne** i zmienne **tymczasowe**,
to **sekcja danych** zawierająca zmienne globalne.

Procesy systemu operacyjnego wykonują kod systemowy, procesy użytkowe kod użytkowników.

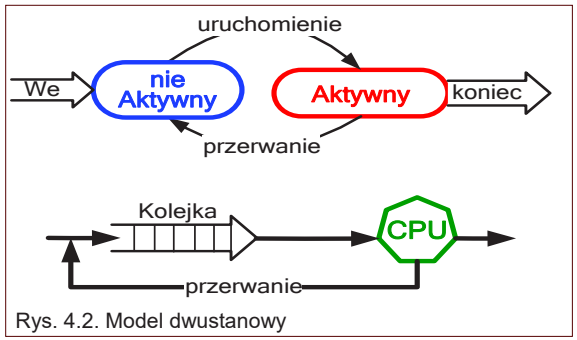
Dwa procesy związane z jednym programem, traktowane będą jako dwie oddzielne sekwencje wykonania.

Przerwania: zdarzenia, które są niezależne od aktualnie działającego procesu i występują na zewnątrz w stosunku do niego (zakończenie We/Wy).
Pułapka: zdarzenia wywołane wystąpieniem błędów lub zdarzeń wyjątkowych wewnątrz aktualnie realizowanego procesu (nielegalna próba dostępu do pliku).

4.1. Modele procesów

❐ Dwustanowy model procesu

Proces może być wykonywany przez CPU lub **nie.**



Rys. 4.2. Model dwustanowy

Dopuszczalne są dwa stany procesu:

„**AKTYWNY**" lub „**nieAKTYWNY**".

Proces „**nieAKTYWNY**" jest widziany przez SO i oczekuje na uruchomienie w kolejce procesów oczekujących.

Co pewien czas proces przetwarzany przez CPU zostaje **przerwany** i SO wybiera - za pomocą **dyspozytora** - następny proces do uruchomiona.

Zatrzymany proces przechodzi w stan „**nieAKTYWNY**" i trafia do **kolejki** a proces wybrany w stan „**AKTYWNY**".

Model dwustanowy jest niewystarczający, gdy spośród **nieDziałających** procesów część jest **zablokowana** z powodu oczekiwania na zakończenie operacji We/Wy.

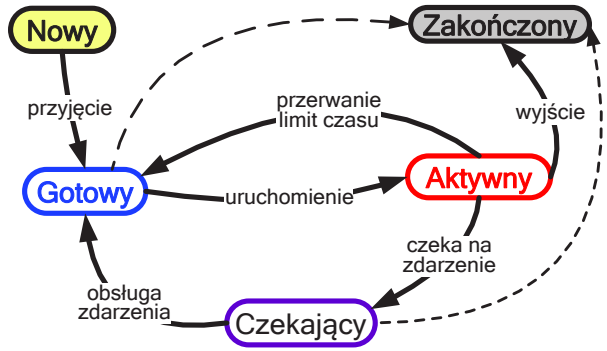
Dyspozytor nie może pobrać ostatniego elementu z kolejki, lecz musi ***wybrać*** z niej proces **nieZablokowany** i najdłużej oczekujący.

➔ Model pięciostanowy

Rozwiązaniem problemu modelu dwustanowego jest podział **nieDziałających** procesów na:

Gotowe do pracy (czekające na CPU)

Zablokowane Czekające na zdarzenie.



Rys. 4.3. Model pięciostanowy

Stany procesu:

NOWY -proces został utworzony, lecz nie jest w grupie gotowych do uruchomienia przez SO;

AKTYWNY -wykonywane są instrukcje programu;

CZEKAJĄCY (**Zablokowany**) -proces czeka na wystąpienie **zdarzenia** (np. na zakończenie operacji We/Wy);

GOTOWY -proces czeka na przydział CPU;

ZAKOŃCZONY -proces zakończył działanie, został usunięty.

W każdej chwili tylko jeden proces może być AKTYWNY, na określonym CPU ale **wiele procesów może być GOTOWYCH do działania** lub **CZEKAJĄCYCH**.

Stan **NOWY** - SO nie podjął decyzji o jego uruchomieniu.

Nowy użytkownik rejestruje się do systemu lub pojawiło się do przetwarzania zadanie wsadowe.

SO: -kojarzy z procesem identyfikator,
-buduje i alokuje w pamięci tablice do zarządzania procesem.

Pozostaje on poza pamięcią główną, w której utrzymywane są tablice z informacjami potrzebnymi do sterowania nim.

Do PAO nie jest przepisany kod programu, i nie został dla niego wydzielony w niej obszar.

Program pozostaje zazwyczaj na dysku.

Stany **NOWY** może być wynikiem ograniczeń nałożonych na liczbę przetwarzanych plików.

Stan **ZAKOŃCZONY** -proces **nie nadaje się** do uruchomienia.

Proces zostaje zakończony wskutek: -zrealizowania swoich zadań,
-wystąpienia błędu,
-zamknięcia przez inny proces o wyższym priorytecie.

Wszelkie tablice i dane związane z zadaniem są jednak tymczasowo przetrzymywane przez SO, aby umożliwić programom pomocniczym pobranie stosownych informacji.

Programy narzędziowe mogą potrzebować danych o przebiegu procesu do analiz wydajności.

Gdy programy pomocnicze zakończą działanie, proces zostaje usunięty z systemu.

AKTYWNY ➔ **GOTOWY** -wyczerpanie limitu czasu, wynikającego z trybu wielozadaniowego.

W systemach z priorytetami procesy o wyższym priorytecie mogą wywłaszczać procesy o niższym priorytecie.

Niech CPU realizuje kod procesu **A**, natomiast proces **B** (o **wyższym** priorytecie) - pozostaje **CZEKAJĄCY**.

Gdy SO otrzyma sygnał o zdarzeniu, na które czekał proces **B**, wówczas proces **A** zostanie zatrzymany i przeniesiony do stanu **GOTOWY**, a sterowanie otrzyma proces **B**.

AKTYWNY ➔ **CZEKAJĄCY** -proces blokuje się gdy zażąda od SO czegoś, na co musi czekać.

Takie żądania mają postać wywołań usług systemowych, rozkazów pochodzących z działającego programu, które odwołują się do procedur systemu operacyjnego.

Proces żąda wykonania usługi SO, której ten nie jest w stanie **natychmiast** wykonać, np.: udostępnienia plików czy współdzielonych obszarów pamięci.

Może też wystąpić oczekiwanie na zakończenie zainicjowanego zadania, jak operacja We/Wy.

Gdy procesy mogą wymieniać informacje między sobą, to blokada może być spowodowana oczekiwaniem na dane lub sygnał od innego programu.

GOTOWY ➔ **ZAKOŃCZONY** - występuje w SO dopuszczających, by proces rodzicielski mógł w dowolnym momencie zamknąć proces potomny.

Czasami zakończenie procesu rodzicielskiego może spowodować zamknięcie wszystkich jego procesów potomnych

NOWY ➔ **GOTOWY** -gdy SO przygotowany jest do obsługi nowego procesu.

Istnieją ograniczenia dotyczące liczby obsługiwanych procesów lub objętości pamięci wirtualnej, którą można dla nich alokować.

4.2. Struktura procesu

Proces opisuje struktura umożliwiającą: -śledzenie jego przebiegu,
-udostępnianie aktualnego stanu,
-lokalizację w pamięci.

Blok **K**ontrolny **P**rocesu jest najważniejszą strukturą danych w systemie operacyjnym.

❒ **Blok kontrolny procesu** (***P**rocess **C**ontrol **B**lock* - **PCB**) reprezentuje proces i zawiera:

-dane identyfikujące proces;
identyfikatory: procesu, procesu rodzicielskiego; użytkownika;

-stan procesu;
licznik rozkazów -adres następnego rozkazu do wykonania w procesie;
rejestry procesora;

-informacje o planowaniu przydziału procesora -priorytet procesu, wskaźniki do kolejek porządkujących zamówienia, inne parametry planowania;

-informacje o zarządzaniu pamięcią -zawartości rejestrów granicznych, tablice stron;

-informacje do rozliczeń -ilość zużytego czasu procesora i czasu rzeczywistego, numery kont, numery procesów itp.;

-informacje o stanie We/Wy -wykaz otwartych plików itd.

Informacje o stanie **rejestrów** i **licznika rozkazów** są przechowywane podczas przerwań, aby proces mógł być później kontynuowany.

Czym jest proces w sensie fizycznym ?

Programem lub rodziną programów do uruchomienia.

Programy to komórki pamięci zawierające zmienne lokalne, globalne oraz zdefiniowane stałe.

Proces to duży obszar pamięci na programy, ich stosy i dane sterujące związane z procesem.

Obraz procesu: zbiór, składający się z programów, danych, stosu i atrybutów.

Obraz procesu może składać się z kilku **bloków**, które nie muszą zajmować spójnego obszaru pamięci.

Bloki mogą mieć ustaloną długość (strony) lub być różnej długości (segmenty).

Fragment obrazu procesu **musi** znajdować się w PAO aby SO mógł zarządzać procesem (podczas gdy reszta na dysku).

Tablice Procesów muszą wskazywać lokalizację każdej strony/segmentu wszystkich obrazów procesów.

Uruchomienie procesu wymaga umieszczenia w PAO (lub pamięci wirtualnej) obrazu procesu.

SO musi znać: -położenie każdego procesu na dysku

-lokalizację wszystkich procesów zapisanych w PAO.

Kiedy proces jest przenoszony do obszaru wymiany, część jego obrazu może pozostać w PAO.

SO musi kontrolować, jaki fragment obrazu każdego procesu pozostaje w pamięci głównej.

System operacyjny musi znać położenie procesu oraz parametry niezbędne do zarządzania nim, (identyfikator procesu, stan, alokacji w pamięci), lub co najmniej wskaźnik do jego obrazu.

- **Przełączanie kontekstu** *(context switch)* –zmiana przydziału CPU :

  -przesłanie do **jądra** informacji o starym procesie,

  -załadowanie przechowywanej w **jądrze** informacji o procesie planowanym do uruchomienia.

Kontekst tworzą: rejestry CPU, stan procesu, informacje dotyczące zarządzania pamięcią.

Podczas przełączania system nie wykonuje żadnej użytecznej pracy.

➔ Przy dużej liczbie zbiorów rejestrów, przełączenie kontekstu sprowadza się do zmiany wartości wskaźnika bieżącego zbioru rejestrów.

Złożoność SO zwiększa nakład pracy podczas przełączania.

> Przygotowując pamięć dla następnego zadania, należy przechować **przestrzeń adresową** bieżącego procesu.

Przełączanie kontekstu jest wąskim gardłem systemu operacyjnego.

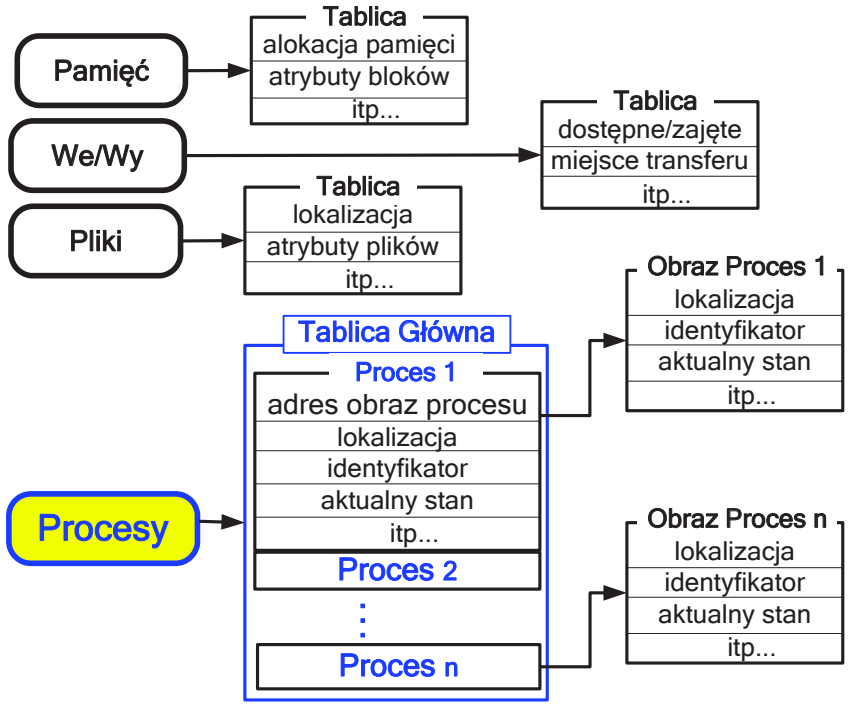
- System operacyjny dysponuje **Tablicami** do **zarządzania procesami**.

Dostępne są **cztery** typy tablic, które są ze sobą powiązanie.

Pamięć RAM, urządzenia **We/Wy** oraz **Pliki** zarządzane są w ***imieniu procesów***, więc odwołania do nich znajdują się pośrednio lub bezpośrednio w **Tablicy Procesów**.

Tablice plików odzwierciedlają, że pliki dostępne są za pośrednictwem urządzeń We/Wy i bywają zapisywane w pamięci głównej lub wirtualnej.

☛ Same Tablice podlegają mechanizmom zarządzania pamięcią ze względu na ciągły dostęp do nich przez system operacyjny.



Rys. 4.6. Tablice sterujące SO

➔ Skąd system operacyjny pobiera informację do wypełniania Tablic ?

Musi znać podstawowe parametry środowiska (ilość dostępnej pamięci, liczbę, rodzaj oraz identyfikatory urządzeń We/Wy, itp.), które zależne są od konfiguracji systemu.

Dane te muszą być zgromadzone poza systemem operacyjnym lub przez oprogramowanie weryfikujące konfigurację i dostarczone systemowi w czasie jego **inicjacji**.

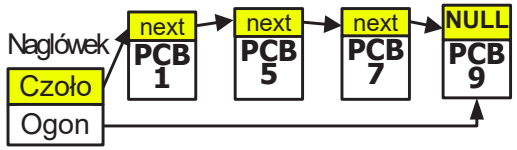
4.3. Planowanie procesów

W systemie jednoprocesorowym aktywny może być tylko jeden proces.

Inne procesy, muszą czekać aż CPU będzie wolne.

Kolejka zadań *(job queue)*: wszystkie procesy w systemie.

Kolejka gotowych procesów: gotowe do działania procesy oczekujące w PAO**.**



Rys. 4.7. Kolejka gotowych procesów.

Kolejka jest listą wiązaną:

-nagłówek zawiera wskaźniki do pierwszego i ostatniego bloku kontrolnego procesu na liście.

Każdy PCB ma pole wskazujące następną pozycję w kolejce procesów Gotowych.

Każde urządzenie ma własną kolejkę.

Kolejka do urządzenia *(device queue)*: -lista procesów czekających na konkretne urządzenie.

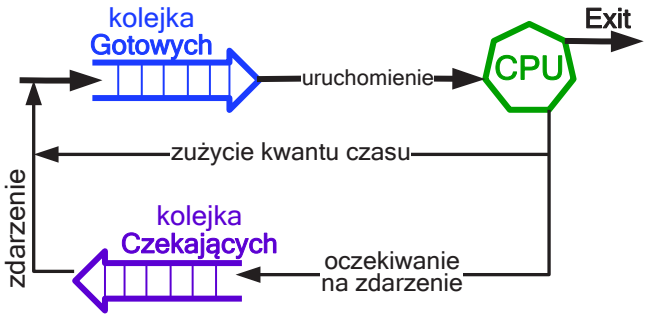
Diagram kolejek jest narzędziem ułatwiającym planowanie (szeregowanie) procesów.

W diagramach występują 2 lub 3 typy kolejek: -procesów **Gotowych**,
-procesów **Czekających**
-kolejka do **urządzeń**.

Prostokąt przedstawia kolejkę.

Kółka oznaczają zasoby obsługujące kolejki.

Strzałki pokazują kierunek przepływu procesów.



Rys. 4.8a. Diagram z dwoma kolejkami

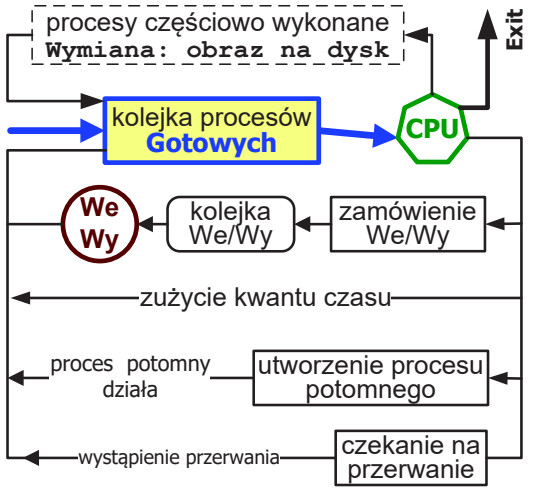
➔ Nowy proces trafia do kolejki procesów **Gotowych**.
Oczekuje w niej do czasu, aż zostanie wybrany do wykonania i otrzyma CPU

Proces który otrzymał CPU może:

1. **zamówić** operację We/Wy, wskutek czego trafia do kolejki procesów **Oczekujących**;
2. **utworzyć** nowy podproces i oczekiwać na jego zakończenie;
3. zostać przymusowo **usunięty** (przerwanie) i przeniesiony do kolejki procesów **Gotowych**.

W przypadkach (1) i (2) proces zostanie w końcu przełączony ze stanu Oczekiwania do stanu Gotowości i przeniesiony do kolejki procesów **Gotowych**.

W tym cyklu proces kontynuuje działanie aż do zakończenia, po czym usuwa się go ze wszystkich kolejek i zwalnia jego blok kontrolny oraz przydzielone mu zasoby.



Rys. 4.8b. Diagram planowania procesów.

Proces wędruje między różnymi kolejkami przez cały czas swego istnienia.

Proces systemowy planista *(scheduler)* **wybiera procesy z kolejek.**

Często występuje więcej procesów niż można ich *"natychmiast"* wykonać; są przechowywane w pamięci **masowej**, gdzie oczekują na późniejsze wykonanie.

Planista krótkoterminowy (*shortterm scheduler)* wybiera jeden proces z procesów **Gotowych** do wykonania i przydziela mu **CPU**.

Musi często wybierać nowy proces dla CPU.

Proces może działać kilka milisekund, a potem przejść w stan **Oczekiwania**, po złożeniu zamówienia na operację We/Wy.

Planista krótkoterminowy musi być bardzo szybki

Planista długoterminowy (*longterm scheduler)* wybiera procesy z pamięci **masowej** i ładuje je do pamięci w celu wykonania.

Planista **długoterminowy**: -działa rzadziej, między kolejnymi procesami mogą upływać minuty.
-nadzoruje **stopień wieloprogramowości** - liczba procesów w pamięci.
-może być wywoływany wtedy, gdy jakiś proces opuszcza system.

Stabilny stopień wieloprogramowości: średnia liczba utworzonych procesów równa się średniej liczbie procesów usuwanych z systemu.

Dłuższe przerwy między wykonaniami dają planiście więcej czasu na rozstrzyganie, który proces należy wybrać do wykonania.

Wybory dokonane przez planistę długoterminowego mają znaczenie strategiczne.

〈 Procesy dzieli się na: $P_{We/Wy}$ -ograniczone przez We/Wy,
P_{Cu} -ograniczone przez dostęp do procesora.

Proces ograniczony przez **We/Wy** spędza większość czasu na wykonywaniu operacji We/Wy.

Proces ograniczony przez **dostęp** do **procesora** sporadycznie generuje zamówienia na We/Wy, spędzając czas na obliczeniach wykonywanych przez procesor.

Planista długoterminowy powinien dobrać mieszankę procesów zawierającą procesy ograniczone przez We/Wy, jak i ograniczone przez dostęp do procesora.

Jeśli wszystkie procesy ograniczone są przez **We/Wy**, to kolejka procesów **Gotowych** będzie prawie zawsze pusta i planista krótkoterminowy będzie miał za mało do roboty.

Jeśli wszystkie procesy są ograniczone **przez dostęp do CPU**, to kolejka do **urządzeń We/Wy** będzie prawie zawsze pusta i system nie będzie zrównoważony.

Planista długoterminowy może być nieobecny lub **zredukowany.**

Systemy z podziałem czasu często nie mają planisty długoterminowego i umieszczają każdy nowy proces w pamięci pod opieką planisty krótkoterminowego.

Stabilność tych systemów zależy od fizycznych ograniczeń (jak liczba dostępnych terminali) lub od zdolności adaptacyjnych użytkujących je ludzi.

W niektórych SO może występować planista średnioterminowy.

Czasami **usuwa się procesy** z PAO (i z aktywnej rywalizacji o CPU) w celu zmniejszenia stopnia wieloprogramowości.

Usunięte procesy można później wprowadzić do pamięci, i kontynuować ich wykonanie od miejsc, w których je przerwano.

Postępowanie takie nazywa się **wymianą** (*swapping).*

4.4. Działania na procesach

Procesy w systemie mogą być wykonywane współbieżnie oraz **dynamicznie** tworzone i usuwane.

❒ **Tworzenie procesu**

Proces może tworzyć nowe procesy za pomocą wywołania systemowego **UtwórzProces()**.

W systemie Windows będzie to funkcja systemowa **CreateProcess()** .

Proces macierzysty *(parent process)* oraz **potomkowie** *(children)* czyli utworzone przez niego nowe procesy.

Każdy nowy Proces może tworzyć kolejne procesy - **drzewo procesów**.

Proces potrzebuje zasobów: -czas CPU, -pamięć operacyjna,
-pliki, -urządzeń We/Wy.

Podproces może otrzymać swoje zasoby:

- bezpośrednio od systemu operacyjnego;
- stanowią podzbiór zasobów procesu macierzystego (ogranicza rozmnażaniu procesów).

Niektóre zasoby procesu macierzystego (pamięć, pliki) mogą potomkowie użytkować wspólnie.

\+ Do procesu potomka mogą dotrzeć dane wejście określone przez jego twórcę.

Kiedy powstał proces **potomny** to proces **macierzysty**:

1. **kontynuuje** działanie **współbieżnie** ze swoimi potomkami,
2. **oczekuje** na **zakończenie** działań niektórych lub wszystkich swoich procesów potomnych.

Przestrzeń adresowa nowego procesu:

- proces potomny staje się kopią procesu macierzystego –ułatwia to komunikowanie się procesów,
- proces potomny otrzymuje nowy program -załadowanie do PAO nowego pliku binarnego.

❒ **Czynności prowadzące** do **utworzenia procesu**

1. **Utworzenie identyfikatora procesu.**
   W głównej Tablicy Procesów zostaje założona nowa pozycja.
2. **Alokacja przestrzeni adresowej** dla **procesu**.
   Pamięć potrzebna na programy i dane oraz stosu użytkownika (mogą być wyznaczone domyślnie lub przez użytkownika podczas tworzenia zadania).
   Dla procesu potomnego, proces rodzicielski przekazuje niezbędne wartości w ***parametrze polecenia*** **UtwórzProcesu**().
   Muszą powstać odpowiednie powiązania do istniejących obszarów współdzielonych.
   Na koniec SO alokuje przestrzeń przeznaczoną dla bloku kontrolnego procesu.
3. **Inicjalizacja Bloku Kontrolnego Procesu.**
   Większość elementów o stanie procesora jest inicjowana z wartościami zero.
   Licznik rozkazów, ustawiany na pozycję początku programu, wskaźnik stosu systemowego określa granice stosu związanego z procesem.
   Elementy sterujące procesem inicjowane są standardowymi wartościami domyślnymi.
4. **Ustawienie odpowiednich połączeń.**
   Jeśli SO wykorzystuje do szeregowania zadań **kolejki implementowane jako listy**, wówczas wskaźnik do nowego procesu musi trafić do odpowiedniej kolejki.
5. **Tworzenie pomocniczych struktur danych.**
   Dla każdego procesu SO może utrzymywać plik rozrachunkowy, przechowujący dane potrzebne do różnorodnych rozliczeń i tworzenia statystyk wydajności.

❒ Zakończenie procesu

Muszą istnieć mechanizmy sygnalizowania zakończenia procesu.

Zadanie wsadowe powinno zawierać rozkaz, który generuje przerwania informującego SO.

Dla aplikacji interaktywnych koniec procesu wskazuje odpowiednie działanie użytkownika.

1. Proces kończy się i **jest usuwany za pomocą wywołania funkcji Exit.**

Proces potomny może przekazać dane (wyjście) do procesu macierzystego.

Wszystkie zasoby procesu, w tym pamięć fizyczna i wirtualna, otwarte pliki i bufory We/Wy zostają odebrane przez system operacyjny.

2. Proces może zakończyć inny proces za pomoc funkcji systemowej (np. Abort).

Funkcję tę może wywołać tylko ***przodek*** procesu, który ma być zakończony; w przeciwnym razie użytkownicy mogliby likwidować sobie wzajemnie dowolne zadania.

Proces macierzysty musi znać identyfikatory swoich potomków.

Gdy jakiś proces tworzy nowy proces, wówczas identyfikator nowo utworzonego procesu jest przekazywany do procesu macierzystego.

3. Proces macierzysty może zakończyć swój proces potomny z **różnych przyczyn:**

- potomek nadużył któregoś z przydzielonych mu zasobów, proces macierzysty musi mieć mechanizm sprawdzania stanu swoich potomków;
- wykonywane przez potomka zadanie stało się zbędne;
- proces macierzysty kończy się, a wówczas SO nie pozwala **potomkowi** na dalsze działanie. System operacyjny inicjuje zakończenie wszystkich jego potomków -*kończenie kaskadowe.*

❒ Przyczyny przerwania wykonywania procesu

1. Polecenie administracyjne -bezpośrednie żądanie wywołania funkcji SO

Gdy np.: w trakcie wykonywania procesu zostanie uruchomiony rozkaz wykonania operacji We/Wy (otwarcia pliku).

Wywołanie podobnego rozkazu przekazuje sterowanie do procedury obsługi urządzenia We/Wy, będącej **częścią kodu SO**.

Na ogół polecenia administracyjne powodują przeniesienie procesu do stanu ***Zablokowany***.

2. Przerwanie -zewnętrzne zdarzenie w stosunku aktualnie wykonywanych czynności

- **zegarowe** -system operacyjny decyduje, kiedy aktualnie przetwarzany proces wyczerpie cały limit czasu przeznaczony na bieżący etap jego działania.

  Gdy to nastąpi, proces musi zostać przełączony w stan ***Gotowy***, a SO wyznacza do pracy kolejny program;

- **We/Wy** -system operacyjny analizuje zachodzące operacje We/Wy.

  Jeśli operacja osiągnie stan odpowiadający zdefiniowanemu zdarzeniu, SO przenosi oczekujące na to zdarzenie procesy ze stanu ***Zablokowany*** do stanu ***Gotowy***.

  Następnie SO rozstrzyga, czy **wznowi** działanie aktualnie przetwarzanego procesu, czy ***wywłaszczy*** go, na rzecz, procesu o wyższym priorytecie;

- **nietrafione odwołania do pamięci** -CPU natrafia co jakiś czas na odwołania do adresów pamięci wirtualnej odnoszące się do słów, których nie ma w PAO.

  SO musi wtedy przenieść odpowiedni blok danych z pamięci dyskowej do PAO.

  Po uruchomieniu operacji We/Wy, SO może przenieść **bieżący** proces w stan ***Zablokowany*** i wznowić przetwarzanie innego procesu.

  Kiedy żądany blok danych znajdzie się w PAO, przerwany proces zostanie przeniesiony do stanu ***Gotowy***.

3. Pułapka -problem aktualnie wykonywanych czynności

System operacyjny sprawdza, czy błąd lub warunek wyjątku mają charakter **krytyczny**.

Jeśli tak, aktualnie wykonywany proces zostaje przeniesiony w stan ***Zakończony***, a sterowanie przełączone do innego procesu.

Jeżeli nie, to dalsze działanie systemu SO zależy od jego budowy i charakteru błędu.

Mogą być uruchomione procedury odtworzeniowe, lub system tylko poinformuje użytkownika odpowiednim komunikatem.

SO może uruchomić inny proces lub wznowić wykonywanie przerwanego.

❒ **Przełączanie trybów** (użytkownik/jądro)

Gdy wystąpi **przerwanie** to CPU wykonuje:

1. Zapisuje **kontekst** dotychczas wykonywanego programu (procesu).
2. W liczniku rozkazów ustawia adres początkowy **programu** obsługi **przerwania**.
3. Przełącza przetwarzanie z trybu **użytkownika** na tryb **jądra** (obsługa przerwania może wykonywać uprzywilejowane rozkazy).
4. Pobiera pierwszy rozkaz programu **obsługi przerwania**.

Program obsługi przerwania jest na ogół bardzo krótki i wykonuje kilka podstawowych zadań związanych z przerwaniem.

➔Wystąpienie przerwania nie oznacza natychmiastowego przełączania.

Po wykonaniu procedury obsługi przerwania sterowanie może powrócić do **dotychczas** wykonywanego **procesu** (do wznowienia jego pracy wystarczy informacja o stanie procesu w chwili wystąpienia przerwania).

Na ogół zapisywanie i odtwarzanie tych danych jest realizowane na poziomie sprzętowym.

❒ **Przełączanie procesów**

Przełączanie procesów może nastąpić w dowolnym momencie (gdy system operacyjny przejmie sterowanie od aktualnie działającego procesu).

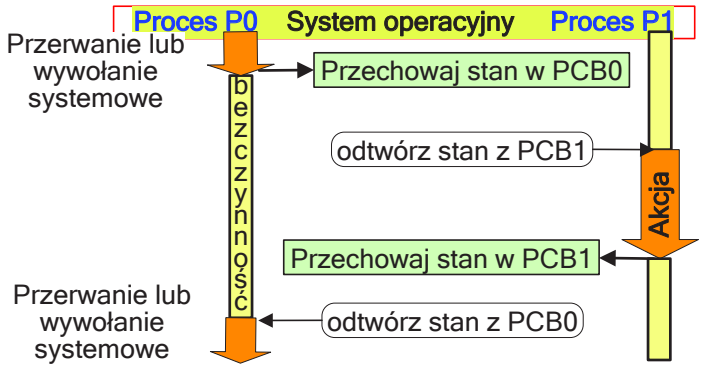
- Należy zdefiniować **zdarzenia** powodujące konieczność przełączania procesów.
- Należy zbudować mechanizm **rozpoznawania**, czy chodzi o przełączanie ***trybów***, czy o przełączanie **procesów**.
- Należy rozwiązać problem **zapisywania** i **odtwarzania** różnorodnych struktur danych służących do sterowania przełączanymi procesami.

Przełączanie procesów wiąże się ze zmianą stanu i obciąża system bardziej niż przełączanie trybów przetwarzania.

Przełączanie trybów (użytkownik/jądro) może zachodzić niezależnie od stanu aktualnie wykonywanego procesu.

Gdy bieżący proces przenoszony jest do innego stanu *(Gotowy, Zablokowany* itp.), wówczas SO dokonuje **znaczących** zmian w środowisku:

1. Zachowuje kontekst CPU (licznik rozkazów, inne rejestry, itp.).
2. Aktualizuje blok sterowania aktualnie wykonywanego procesu (zmiana stanu procesu, nadanie stosownych wartości innym polom, np.: o przyczynie przerwania procesu, dane do rozliczeń).
3. Przenosi Blok Sterowania Procesem do odpowiedniej kolejki *(Gotowy, Zablokowany-czekający).*
4. Wybiera do przetwarzania inny proces.
5. Aktualizuje Blok Sterowania nowo wybranego procesu (zmienić stan procesu na *Działający).*
6. Aktualizuje struktury danych zarządzające pamięcią.
7. Odtwarza kontekst CPU do stanu, jaki miał miejsce przed zatrzymaniem uruchamianego procesu.



Rys. 4.9. Przełączania procesów

4.5. Procesy współpracujące

Procesy współbieżne wykonywane w SO mogą: -być niezależne,
-ze sobą współpracować.

Proces **niezależny** *(independent):* nie może oddziaływać na inne procesy wykonywane w systemie, a te nie mogą wpływać na jego działanie.

Nie dzieli żadnych danych (tymczasowych lub trwałych) z innym procesem.

Proces **współpracujący** *(cooperating):* może wpływać na inne procesy w systemie lub inne procesy mogą oddziaływać na niego.

Proces dzielący dane z innymi procesami jest procesem współpracującym.

Korzyści ze współpracy procesów:

dzielenie informacji: kilku użytkowników może korzystać z tych samych informacjami (wspólne pliki), należy zapewnić współbieżny dostępu do tych zasobów.

przyspieszanie obliczeń: można zadanie podzielić na podzadania, z których każde będzie wykonywane równolegle z pozostałymi.

modularność: konstruowanie systemu w sposób modularny, dzieląc go na osobne procesy.

wygoda: indywidualny użytkownik może mieć wiele zadań do wykonania w jednym czasie; np.: równolegle redagować, drukować i kompilować.

Współbieżność wymaga współpracy między procesami i dostępu do mechanizmów, umożliwiających procesom wzajemne komunikowanie się i synchronizowanie działań.

❒ **Zagadnienie Producent-Konsument** *(Producer-Consumer problem).*

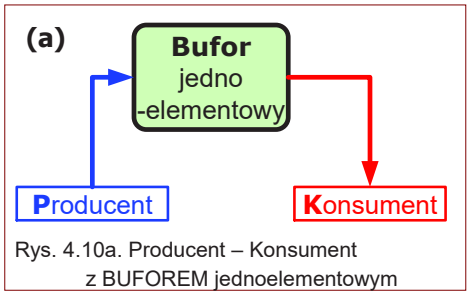
Proces **producent** wytwarza informacje, które zużywa proces **konsument**.
Współbieżne działanie wymaga dysponowania BUFOREM jednostek.

Gdy **producent** tworzy pewną jednostkę, **konsument** może zużywać inną.

Producent wytwarza produkt, umieszcza go w BUFORZE i rozpoczyna pracę od nowa.
W tym samym czasie **konsument** pobiera produkt z BUFORA.

Procesy producenta i konsumenta muszą podlegać **synchronizacji**, aby konsument nie próbował konsumować jednostek, które nie zostały jeszcze wyprodukowane.

Konsument musi czekać na wyprodukowanie tego, co chce konsumować.

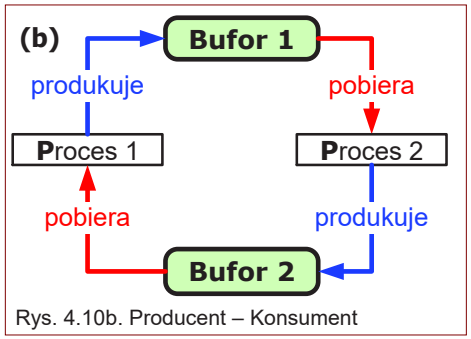


Rys. 4.10a. Producent – Konsument z BUFOREM jednoelementowym

Synchronizacja procesów na gwarantować, że **producent** nie będzie dodawać nowych jednostek gdy BUFOR jest pełny, a **konsument** nie będzie pobierać gdy BUFOR jest pusty.

Problem z **nieograniczonym buforem** *(unbounded-buffer)*: nie ma ograniczeń na rozmiar bufora.

Producent może produkować nieustannie, zaś **konsument** musi **czekać** na nowe jednostki.



Rys. 4.10b. Producent – Konsument

Problem z **ograniczonym buforem** *(bounded-buffer)*: zakłada się, że bufor ma ustaloną długość.

Konsument czeka, gdy bufor jest pusty, a **producent** czeka, jeśli bufor jest pełny.

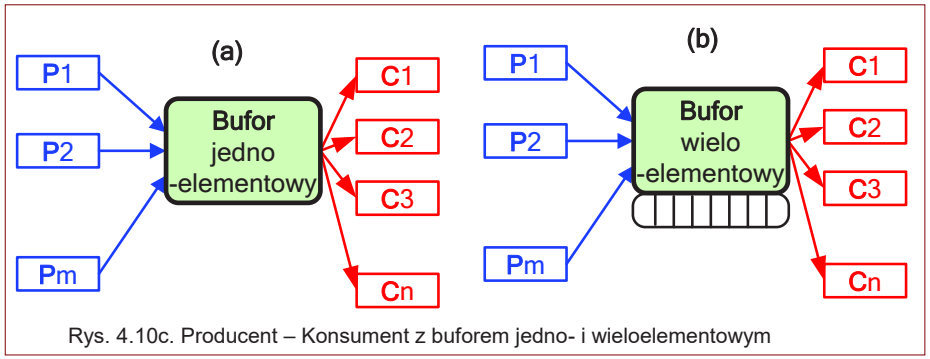
Bufor może być: -jawnie zakodowany z wykorzystaniem pamięci dzielonej.
-udostępniony przez SO za poprzez komunikację międzyprocesową

Generalne rozwiązanie problemu:

Producent: uśpić GO w momencie gdy BUFOR jest pełny.
Pierwszy **konsument**, który pobierze coś z BUFORA budzi **producenta**, który uzupełnia BUFOR

Konsument: uśpić GO w momencie gdy próbuje pobrać z pustego BUFORA.
Pierwszy **producent** dodając nowy produktu do BUFORA budzi konsumenta do działania.

➔Uwaga: Nieprzemyślane rozwiązania mogą powodować **blokadę** procesów (zakleszczenie)←



Rys. 4.10c. Producent – Konsument z buforem jedno- i wieloelementowym

❒ **Podsumowanie**

W systemie działa **P>0** procesów (Producentów), które produkują pewne dane oraz **K>0** procesów (Konsumentów), które odbierają dane od producentów.

Praca Producentów i Konsumentów powinna być synchronizowana, aby:

1. Konsument oczekiwał na pobranie danych gdy BUFOR jest pusty.
2. Producenta wstrzymał umieszczanie danych gdy BUFOR jest pełny.
3. Jeśli wielu Konsumentów oczekuje, aż w BUFORZE pojawią się dane oraz ciągle są produkowane nowe dane, to każdy oczekujący Konsument zawsze coś z bufora pobierze.
   - ☛ Nie może jakiś Konsument czekać w nieskończoność na pobranie danych, które ciągle napływają do BUFORA.
4. Jeśli wielu Producentów oczekuje na wolne miejsce w BUFORZE, a konsumenci ciągle coś z BUFORA pobierają, to każdy oczekujący Producent będzie mógł coś włożyć do BUFORA.
   - ☛Nie może jakiś Producent czekać w nieskończoność, jeśli z BUFORA ciągle coś jest pobierane.
5. Wykluczyć jednoczesne działanie Producenta i Konsumenta w tym samym miejscu BUFORA

Warianty problemu:

1. Może nie być bufora.
2. Bufor cykliczny może mieć ograniczoną pojemność.
3. Bufor może być nieskończony.
4. Może być wielu producentów lub jeden.
5. Może być wielu konsumentów lub jeden.
6. Dane mogą być produkowane i konsumowane po kilka jednostek na raz.
7. Dane muszą być odczytywane w kolejności ich zapisu lub nie.

❒ **Szkic rozwiązania problemu z ograniczonym buforem** i z użyciem **pamięci dzielonej**

Bufor dzielony zrealizowany jest jako tablica cykliczna.

we -wskazuje następne miejsce **wolne** w BUFORZE

wy -wskazuje pierwsze miejsce ***zajęte*** w BUFORZE

BUFOR jest pusty, gdy **we = wy**

BUFOR jest pełny, gdy **we + 1 mod n = wy**

dataP - jednostka nowo produkowana;

dataK -jednostka do skonsumowania.

```
// proces producenta
repeat
      …
  Produkuj(jednostkę dataP)
      …
  while we + 1 mod n = wy do NIC
  BUFOR[we] ← dataP
  we ← we + 1 mod n
until false
```

```
// proces konsumenta
repeat
  while we = wy do NIC
  dataK ← BUFOR[wy];
  wy ← wy + 1 mod n
      …
  Konsumuj(jednostkę dataK)
      …
until false
```

Pętla **while *warunek* do NIC** -sprawdza warunek dopóki nie będzie fałszywy.

4.6. Komunikacja międzyProcesowa

Komunikacja międzyprocesowa (***I**nter**P**rocess-**C**ommunication* –**IPC**) **to oprogramowanie,** umożliwiające procesom łączność i synchronizowanie działań.

Komunikację najlepiej realizuje się za pomocą przekazywania komunikatów.

System komunikatów umożliwia procesom wzajemną komunikację bez zmiennych dzielonych.

Podstawowe operacje: **Nadaj***(komunikat)* i **Odbierz***(komunikat).*

➔ Komunikaty wysyłane przez proces mogą mieć **stałą** lub **zmienną** długość.

Komunikowanie się procesów wymaga **łącza komunikacyjnego** (*communication link).*

Problemy związane z implementacją łącza:

- Czy łącze może być powiązane z więcej niż dwoma procesami?
- Ile może być łączy między każdą parą procesów?
- Jaka jest pojemność łącza i czy łącze ma obszar buforowy?
- Jaki jest rozmiar komunikatów (zmiennej czy stałej długości)?
- Czy łącze jest jednokierunkowe, czy dwukierunkowe?

Łącze jednokierunkowe: każdy podłączony do niego proces może albo nadawać albo odbierać, (nie może wykonywać obu czynności na przemian); każde łącze ma przynajmniej jeden proces odbiorczy.

Metody logicznej implementacji łącza i operacji **Nadaj/Odbierz:**

- komunikacja bezpośrednia lub pośrednia;
- komunikacja symetryczna lub asymetryczna;
- buforowanie automatyczne lub jawne;
- wysyłanie na zasadzie tworzenia kopii lub odsyłacza;
- komunikaty stałej lub zmiennej długości.

Procesy zwracają się do siebie za pomocą komunikacji: **bezpośredniej** lub **pośredniej**.

4.6.1. Komunikacja bezpośrednia

Proces komunikujący się musi **jawnie** nazwać odbiorcę lub nadawcę.

Nadaj*(**P**, komunikat)* -nadaj komunikat do procesu **P**;

Odbierz*(**Q**, komunikat)* -odbierz komunikat od procesu **Q**

Własności łącza komunikacyjnego:

- ustanawiane jest automatycznie między parą procesów (procesy znają swoje identyfikatory);
- łącze dotyczy dokładnie dwu procesów;
- między każdą parą procesów istnieje dokładnie jedno łącze;
- łącze może być jednokierunkowe (zazwyczaj dwukierunkowe).

❒ Problem Producent - Konsument

Producent wytwarza pewną jednostkę, podczas gdy **Konsument** zużywa inną jednostkę.

Gdy **Producent** skończy wytwarzanie jednostki, wysyła ją konsumentowi poprzez operację **Nadaj.**

Konsument pobiera jednostkę za pomocą operacji **Odbierz.**

Jeśli jednostka nie została wytworzona, proces konsumenta musi zaczekać na jej wytworzenie.

```
// proces producenta
repeat
  Wytwarzaj(jednostkę w DATA)
  Nadaj(Konsument, DATA);
until false;
```

```
// proces konsumenta
repeat
  Odbierz(Producent, DATA);
  Konsumuj(jednostkę z DATA)
until false;
```

Schemat wykazuje symetrię adresowania:

proces **nadawczy** i **odbiorczy** muszą wzajemnie ***używać nazw*** aby utrzymać ze sobą łączności

Asymetryczne adresowanie: tylko **Nadawca** nazywa Odbiorcę.

Operacje **Nadaj** i **Odbierz** określa się następująco:

Nadaj(*P*, ***komunikat***) - nadaj komunikat do procesu *P*;

Odbierz(**idn**, ***komunikat***) -odbierz komunikat od **dowolnego** procesu;

pod **idn** zostanie podstawiona nazwa procesu, od którego nadszedł komunikat.

Wady: Zmiana nazwy jednego procesu może wymagać weryfikowania definicji innych procesów.

Należy zlokalizować miejsca wystąpień starej nazwy, aby zastąpić nową nazwą.

Sytuacja niepożądana, biorąc pod uwagę niezależną kompilację.

4.6.2. Komunikacja pośrednia

Komunikaty nadawane i odbierane są za pośrednictwem **Skrzynek Pocztowych** (*mailboxes),* nazywanych także **portami** (*ports).*

Abstrakcyjna skrzynka pocztowa jest obiektem.

Każda skrzynka pocztowa ma **jednoznaczną identyfikację.**

☛ Proces może komunikować się z innymi procesami za pomocą **różnych** skrzynek pocztowych.

☛ Dwa procesy mogą komunikować się tylko wtedy, gdy mają one **wspólną** skrzynkę pocztową.

Nadaj(**SP**, *komunikat*) -nadaj komunikat do skrzynki **SP**

Odbierz(**SP**, *komunikat*) -odbierz komunikat ze skrzynki **SP**

Własności łącza komunikacyjnego w komunikacji pośredniej:

- łącze między dwoma procesami jest ustanawiane wtedy, gdy dzielą one skrzynkę pocztową;
- łącze może być związane z więcej niż dwoma Procesami;
- każda para komunikujących się Procesów może mieć kilka różnych łączy, z których każde odpowiada jakiejś skrzynce pocztowej;
- łącze może być jednokierunkowe lub dwukierunkowe.

Procesy **P1**, **P2** i **P3**, mają wspólną skrzynkę pocztową **SP**.

Proces **P1** wysyła komunikat do **SP**.

Procesy **P2** i **P3** kierują jednocześnie do skrzynki **SP** operację **Odbierz**.

Który proces otrzyma komunikat nadany przez **P1** ?

Możliwości rozwiązania problemu trzech Procesów:

1. zezwolić na połączenie tylko między dwoma procesami;
2. pozwolić tylko jednemu procesowi na wykonywanie w danej chwili operacji **Odbierz;**
3. dopuścić, aby sam system wybierał dowolnie proces, do którego dotrze komunikat.

Skrzynka pocztowa może być własnością procesu albo systemu.

◗ Skrzynka należąca do **Procesu** (przypisana lub zdefiniowana jako część procesu), rozróżnia:

-**Właściciela** (może tylko odbierać komunikaty),

-**Użytkownika** (może tylko nadawać komunikaty).

☛ Każda skrzynka ma jednoznacznie określonego **Właściciela**.

Gdy proces będący **Właścicielem** skrzynki pocztowej kończy działanie, skrzynka znika.

Proces, który próbowałby wysyłać komunikaty do zniknięte skrzynki, musi zostać powiadomiony, że skrzynka już nie istnieje (obsługa sytuacji wyjątkowych).

Jak wyznaczyć Właściciela i Użytkowników skrzynki pocztowej ?

Proces może zadeklarować zmienną typu skrzynka pocztowa.

☛ ***Proces deklarujący skrzynkę pocztową staje się jej Właścicielem.***

Każdy inny proces, który zna nazwę tej skrzynki, może zostać jej **użytkownikiem**.

◗ Skrzynka pocztowa należąca do **SO** istnieje bez inicjatywy procesu**.**

☛ Jest niezależna i nie przydziela się jej do żadnego procesu.

System operacyjny dostarcza mechanizmów:

- tworzenia nowej skrzynki i likwidowania istniejące,
- nadawania i odbierania komunikatów za pośrednictwem skrzynki.

Proces zamawiający nową skrzynkę, staje się jej ***Właścicielem*** na zasadzie domyślności.

Początkowo właściciel jest jedynym Procesem, odbierającym komunikaty przez tę skrzynkę.

Przywilej własności może zostać przekazany innym procesom za pomocą funkcji systemowej.

Zwiększa się liczba odbiorców dla skrzynki.

Procesy mogą dzielić Skrzynkę Pocztową w wyniku tworzenia nowych Procesów.

Jeśli Proces **P** utworzy skrzynkę **SP**, *a* następnie utworzy nowy proces **Q**, to **P** i **Q** będą wspólnie korzystać ze skrzynki **SP**.

Procesy mające prawa dostępu do skrzynki, kiedyś kończą działanie; po pewnym czasie skrzynka pocztowa może stać się niedostępna dla żadnego procesu.

System operacyjny powinien odzyskać obszar, który zajmowała skrzynka.

4.6.3. Asynchroniczna komunikacja:

Proces **P** wysyła komunikat do procesu **Q**, przy czym dalsze **jego** działanie może nastąpić dopiero po odebraniu komunikatu.

Proces **P** wykonuje instrukcje:

Nadaj(**Q**, *komunikat);*

Odbierz(**Q**, *komunikat);*

Proces **Q** wykonuje instrukcje:

Odbierz(**P**, *komunikat);*

Nadaj(**P**, "**potwierdzenie**");

4.6.4. Problemy komunikacji międzyprocesowej

Łącze ma określoną pojemność dla komunikatów, mogących w nim czasowo przebywać.

Mówimy o kolejce komunikatów przypisanych do łącza, którą można implementować trojako:

pojemność zerowa: łącze **nie** zezwala, by czekał w nim jakikolwiek komunikat.

Nadawca **musi czekać**, aż Odbiorca odbierze komunikat.

Oba procesy muszą być zsynchronizowane co określa się nazwą ***rendez-vous***.

⇐ Systemem komunikatów bez buforowania.

pojemność ograniczona: może w niej pozostawać co najwyżej **n** komunikatów.

Jeśli w chwili nadania nowego komunikatu kolejka nie jest pełna, to nowy komunikat zostaje w niej umieszczony (skopiowanie komunikatu lub zapamiętanie wskaźnika do niego) i **Nadawca** może kontynuować działanie bez czekania.

☛ Gdy łącze jest zapełnione **Nadawca** czeka, aż zwolni się miejsce w kolejce.

pojemność nieograniczona: może w niej oczekiwać dowolna liczba komunikatów.

Dwie ostatnie metody stosują automatyczne buforowanie.

Dla **niezerowej pojemności** proces nie wie, czy komunikat dotarł do celu po zakończenia operacji **Nadaj**().

Nadawca **musi jawnie** skontaktować się z Odbiorcą, aby sprawdzić czy otrzymał przesyłkę.

❒ **Przypadki specjalne**

Proces nadający komunikat nigdy nie jest opóźniany.

Jeśli **Odbiorca** nie zdąży przyjąć komunikatu, zanim **Nadawca** nie wyśle następnego, to pierwszy komunikat jest tracony.

Procesy wymagają ***jawnej*** synchronizacji, aby:

-żaden z komunikatów nie został zagubiony,

-**Nadawca** i **Odbiorca** nie korzystali jednocześnie z bufora komunikatów.

Proces Nadający komunikat jest **opóźniany** do **czasu otrzymania odpowiedzi.**

Komunikaty tego systemu mają stałą długość (osiem słów).

Proces **P** po nadaniu komunikatu jest wstrzymywany do czasu, aż proces odbiorczy otrzyma komunikat i wyśle ośmiosłowową odpowiedź za pomocą operacji **Odpowiedz**(**P**, *komunikat).*

Komunikat z odpowiedzią zapisuje się w tym samym buforze co komunikat nadany na początku.

Operacja **Nadaj**() powoduje wstrzymywanie procesu nadawczego.

Operacja **Odpowiedz**() pozwala natychmiast kontynuować oba procesy.

Nadawca/Odbiorca kończy działanie przed **zakończeniem przetwarzania komunikatu.**

-Pozostaną komunikaty, których nikt nigdy nie odbierze,

-Jakieś procesy będą czekać na komunikaty, które nigdy nie zostaną wysłane.

1. Proces **P** może **czekać** na komunikat od procesu **Q**, który zakończył działanie.

   Jeśli nie podejmie się żadnych kroków, to proces **P** zostanie ***zablokowany*** na zawsze.

   System może zakończyć proces **P** albo powiadomić go, że proces **Q** zakończył swą pracę.

2. Proces **P** **wysyła** komunikat do procesu **Q**, który już zakończył działanie.

   W systemie **automatycznego** buforowania nie powoduje to szkody - proces **P** kontynuuje działanie.

   Jeśli proces **P** potrzebuje się upewnić, że komunikat został przetworzony przez proces **Q**, to powinien ***jawnie*** poprosić o *potwierdzenie*.

   W przypadku systemu ***bezbuforowego*** proces **P** zostanie zablokowany na zawsze.

   System może zakończyć proces **P**, albo zawiadomić go, że proces **Q** już nie istnieje.

Komunikat nadany przez **proces P** do **procesu Q** **ginie** w **sieci** z **powodu awarii.**

Metody postępowania:

1. SO jest odpowiedzialny za wykrycie takich zdarzeń i za **ponowne** nadanie komunikatu.
2. Proces nadawczy jest odpowiedzialny za wykrycie takiego zdarzenia i powtórne przesłanie komunikatu, jeśli mu na tym zależy.
3. SO odpowiada za wykrywanie takich zdarzeń:
   - zawiadamia on proces nadawczy, że komunikat zaginął.
   - proces nadawczy może postąpić według własnych potrzeb.

4.7. Wątki *(Threads)*

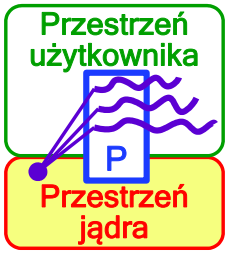
Proces określają używane zasoby i miejsce, w którym działa.

Wykorzystanie zasobów będzie lepsze, gdy będą używane wspólnie i współbieżnie.

4.7.1. Wątki poziomu jądra *(kernel level threads* - **KLT***)*

Wątki **KLT** nazywa się czasami procesami lekkimi *(**L**ight **W**eight **P**rocess* -LWP)

Wątki poziomu jądra zarządzane są **wyłącznie** przez **jądro**, poprzez zbiór stosownych funkcji.



Aplikacja nie zawiera żadnego fragmentu kodu do zarządzania wątkami, a jedynie **I**nterfejs **P**rogramów **U**żytkowych (**API**) do mechanizmów obsługi wątków, dostępnych tylko w **jądrze**.

Każdą aplikację można napisać wielowątkowo.

Wszystkie **wątki** aplikacji obsługiwane są w ramach **jednego** procesu.

Jądro utrzymuje informacje o kontekstach:
-procesu jako całości
-jego pojedynczych wątków.

Jądro szereguje wątki i może:
-równocześnie uruchamiać wiele wątków tego samego procesu na wielu CPU,
-w razie zablokowania jednego wątku procesu uruchomić jego inny wątek.

Wątek to podstawowa jednostka **wykorzystania CPU**, o składzie:
-licznik rozkazów,
-zbiór rejestrów,
-obszar stosu.

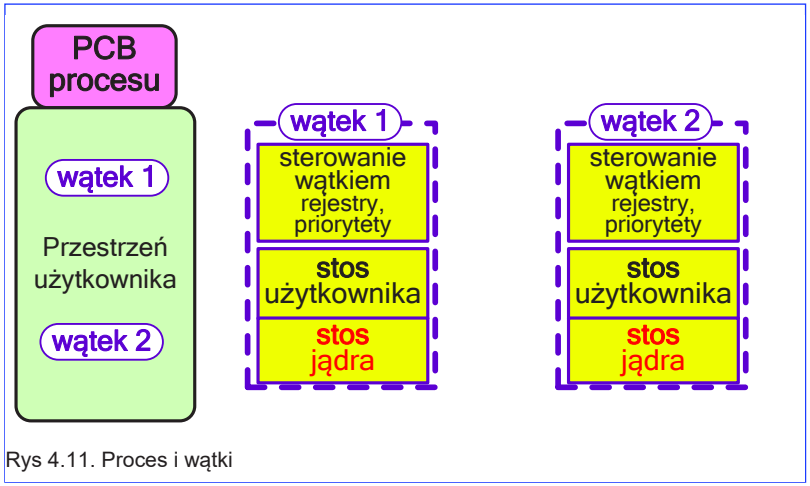
Wątek współużytkuje z równorzędnymi wątkami:
-sekcję kodu,
-sekcję danych,
-otwarte pliki i sygnały, co łącznie stanowi **zadanie** *(task).*

Tradycyjny proces jest równoważny zadaniu z jednym wątkiem.

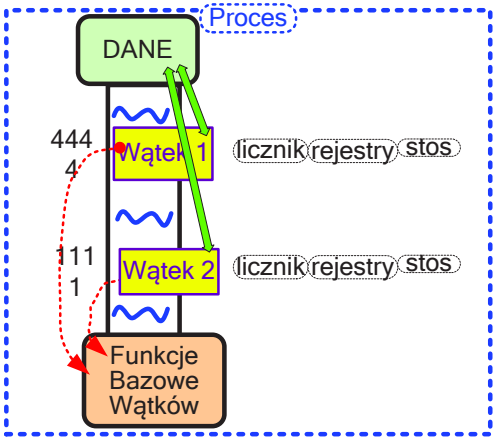
➔ **Proces nic nie robi, jeśli nie ma w nim wątku.**

☛ Wątek może przebiegać w dokładnie jednym Procesie.

Przełączanie CPU między równorzędnymi **wątkami** jest tanie w porównaniu z przełączaniem kontekstu między **procesami**.



Rys 4.11. Proces i wątki



Rys 4.12. Wątki

Przełączanie kontekstu między wątkami wymaga przełączania zbioru rejestrów, jednak nie trzeba wykonywać prac związanych z zarządzaniem pamięcią.

Każdy **proces** działa niezależnie od innych (własny licznik rozkazów, stos i przestrzeń adresowa).

Jest to wygodne **wtedy**, gdy zadania wykonywane przez procesy nie są ze sobą powiązane.

Wiele procesów może wykonywać to **samo zadanie.**

Każdy proces może wykonywać **ten sam** program, ale we własnej pamięci i z własnymi plikami.

☛ **Często wydajniej jest stosować jeden proces z wieloma wątkami.**

Proces wielowątkowy zużywa mniej zasobów niż zwielokrotnione Procesy, biorąc pod uwagę:

-pamięć operacyjną,

-otwarte pliki,

-planowanie procesora.

Uwaga: wątki użytkują wspólnie CPU - w danej chwili tylko **jeden** wątek jest **a**ktywny.

Wykonanie wątku w procesie jest sekwencyjne: każdy wątek ma własny stos i licznik rozkazów.

Stany wątków:

Tworzony -utworzenie nowego procesu oznacza powstanie odpowiadającego mu wątku. Każdy wątek może tworzyć inne wątki tego samego procesu.

Gotowy -nowy wątek zostaje umieszczony w kolejce obiektów gotowych.

Zablokowany -wątek czekający na wystąpienie **zdarzenia**, jest blokowany, co oznacza zapisanie rejestrów użytkownika, licznika rozkazów i wskaźników stosów.

Odblokowany -wystąpienie zdarzenia sygnalizującego zakończenie czynności blokujących, przenosi wątek do kolejki wątków gotowych.

Aktywny -CPU przystępuje do przetwarzania gotowego do działania wątku.

Zakończony –po zakończeniu przetwarzania wątku, zwalniane są obszary pamięci zajmowane przez kontekst jego rejestrów oraz stosów.

Wątki mogą tworzyć wątki potomne.

☛ Mogą blokować się do czasu zakończenia wywołań systemowych.

☛ Jeśli jeden wątek jest zablokowany, to może działać inny wątek.

Blokowanie jednego wątku i przełączanie do innego wątku pozwala na wydajne obsługiwanie przez serwer wielu zamówień.

⇨ **Wątki nie są niezależne od siebie (procesy tak).**

Wątki jednego procesu korzystają z tej samej przestrzeni adresowej jak też innych zasobów (pliki) procesu.

Wszystkie wątki mają dostęp do każdego adresu w zadaniu, mogą czytać i zapisywać stosy innych wątków.

☛ Modyfikacja zasobów przez jeden wątek wpływa na środowisko innych wątków.

☛ Istnieje konieczność synchronizacji działań poszczególnych wątków.

⇨ **Brak ochrony na poziomie wątków.**

Procesy pochodzą od różnych użytkowników ⇐ **mogą kolidować** ze sobą (dlatego są chronione).

Zadanie z wieloma wątkami może należeć do jednego użytkownika, który zadba aby nie przeszkadzały sobie nawzajem.

Implementacja wątków na poziomie jądra umożliwia wielowątkową organizację samego jądra.

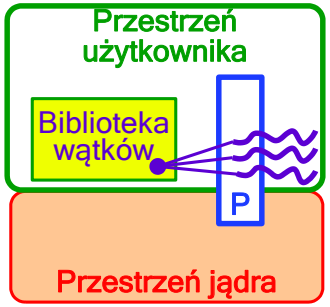
Aplikacje z wątkami poziomu jądra, wymagają przełączania na tryb jądra za <u>każdym razem</u>, gdy przełączane jest sterowania między wątkami tego samego procesu.

4.7.2. Wątki poziomu użytkownika *(User Level Threads* - **ULT***)*

Wykorzystują ***wywołania biblioteczne*** zamiast odwołań do systemu, przełączanie wątków nie wymaga wzywania SO i przerwań związanych z przełączaniem na tryb jądra.

Wątki ULT zarządzane są wyłącznie przez **aplikacje**.

Jądro nie jest odpowiedzialne za ich istnienie.



Korzysta się z biblioteki wątków (procedury zarządzania wątkami), która zawiera kod do ich tworzenia i usuwania, do przesyłania komunikatów i danych między nimi, a także do ***szeregowania*** wątków oraz zapisywania i odtwarzania ich kontekstów.

Aplikacja rozpoczyna pracę z jednym wątkiem.

Aplikacja i jej wątek alokowane są w jednym **procesie** zarządzanym przez **jądro**.

Aplikacja może powołać nowy wątek w ramach tego samego procesu, poprzez wywołanie odpowiedniego **<u>programu narzędziowego</u>** z ***biblioteki wątków***.

Aplikacja przekazuje sterowanie do tego **<u>programu</u>** za pomocą wywołania procedury.

Programy ***biblioteki wątków*** tworzą strukturę danych dla nowego wątku i przekazują sterowanie jednemu z wątków pozostających w stanie gotowości.

Wybór uruchamianego wątku realizuje algorytm szeregowania (z biblioteki wątków).

➔ **Podczas przetwarzania kodu z biblioteki wątków Proces może zostać przerwany** ➔ (wyczerpanie limitu czasu, wywłaszczenie).

W momencie przerwania Proces <u>może realizować przełączanie</u> z jednego **wątku** na drugi.

Z chwilą wznowienia Procesu najpierw dokończone zostanie przełączanie wątków, a następnie przekazanie sterowania do nowego wątku w ramach tego procesu.

❒ **Zalety wątków ULT**

1. Zarządzanie wątkami nie wymaga, by proces przechodził w tryb **jądra**, gdyż wszystkie struktury danych znajdują się w przestrzeni adresowej jednego procesu.
   ☛ Unika się obciążenia powodowanego przejściami z trybu użytkownika do trybu jądra.
2. Aplikacje mogą różnić się od siebie algorytmami szeregowania wątków.
   Dostosowanie algorytmu do potrzeb aplikacji **nie ma** żadnego wpływu na szeregowanie **Procesów** na poziomie systemu operacyjnego.
3. Wątki ULT można stosować w dowolnym SO, gdyż do ich obsługi nie wymagana jest modyfikacja **jądra**.
   Biblioteka wątków zawiera narzędzia dostępne na poziomie aplikacji.

❒ **Wady wątków ULT:**

1. Często wywołania systemowe powodują blokowanie **Procesu**. Gdy wątek ULT zastosuje takie wywołanie, wtedy zablokuje siebie i **cały Proces**, którego jest elementem.
2. Stosowanie wątków ULT w czystej postaci uniemożliwia aplikacjom wielowątkowym korzystanie z zalet przetwarzania wieloprocesorowego.

   Jądro przyznaje w tym samym czasie tylko **jeden CPU** dla **Procesu**;
   nie można przetwarzać jednocześnie kilku wątków **jednego Procesu**.

   Następuje utrata korzyści wynikającej z możliwości współbieżnego wykonywania różnych fragmentów kodu na maszynach wieloprocesorowych.

☛ Wymienione problemy można obejść pisząc aplikacje, które posługują się tylko Procesami, lecz wówczas przełączanie odbywa się między Procesami, co jest kosztowne.

● Technika **osłaniania** *(jacketing)* to sposób ominięcia problemu blokowania wątków.

Stosuje się konwersję wywołania blokującego na nieblokujące.

Watek **nie wywołuje systemowej** procedury obsługi urządzeń We/Wy, lecz analogiczną procedurę **osłonową** na poziomie aplikacji, w której znajduje się fragment kodu **kontrolujący**, czy **urządzenie We/Wy jest zajęte**.

Jeśli tak, wówczas wątek przechodzi w stan ***Gotowości***, za pośrednictwem biblioteki wątków i przekazuje sterowanie innemu wątkowi.

Kiedy sterowanie powróci do wątku wyposażonego w procedurę osłonową, wtedy ten rozpocznie swoją pracę od ponownego sprawdzenia **dostępności** urządzenia We/Wy.

Wątki poziomu użytkownika nie angażują jądra; można je przełączać szybciej niż wątki realizowane przez jądro.

Wywołanie *SO* (np. We/Wy) **może powodować oczekiwanie całego procesu, ponieważ jądro planuje tylko procesy (nie wie o wątkach), a proces czekający nie otrzymuje przydziału czasu procesora.**

Stosując **wątki** poziomu **użytkownika** zamiast **procesów**, czy wątków poziomu **jądra** oczekuje się wzrostu wydajności.

Gdy większość przełączeń zachodzących w trakcie wykonywania programu wymaga dostępu w trybie jądra (np. We/Wy), wówczas konstrukcje oparte na wątkach poziomu użytkownika mogą nie przynieść oczekiwanych korzyści.

P-wątki (Pthreads) to specyfikacja standardu POSIX, który definiuje interfejs programów użytkowych do tworzenia i synchronizacji wątków.

Programy w języku C++ używające P-wątków zawierają plik nagłówkowy <**pthread.h**>, który realizuje implementacje interfejsu **Pthread API** jako biblioteki wątków poziomu użytkownika.

Problem Producent-Konsument, wymaga dzielenia wspólnego BUFORA.
Producent i **Konsument** mogą być wątkami **jednego zadania**.
Przełączanie między nimi będzie tanie.
W systemie dwuprocesorowym oba wątki mogą działać równolegle.

4.7.3. Podsumowanie

Zazwyczaj przeniesienie procesu do obszaru wymiany oznacza przeniesienie tam wszystkich jego wątków, gdyż współużytkują one tę samą przestrzeń adresową.

Przełączanie wątków organizowanych przez jądro zabiera więcej czasu, gdyż zajmuje się tym jądro za pośrednictwem przerwań.

⇨ Planowanie może być niesprawiedliwie.

Proces **A** ma jeden wątek.
Proces **B** ma 100 wątków.
Oba procesy otrzymają tę samą liczbę kwantów czasu.
Wątek w procesie **A** będzie działał 100 razy szybciej niż wątek w procesie **B**.

⇨ **Każdy wątek może** jednak **podlegać indywidualnemu planowaniu**.

Wówczas proces **B** otrzyma 100 razy więcej czasu procesora niż proces **A**.

Proces **B** wykonujący np. współbieżnie **100** funkcji systemowych, może zrobić **więcej** niż taki sam działający w systemie z wątkami poziomu użytkownika.

🕮 Koncepcje systemu operacyjnego 🕮

❒ **System operacyjny** jako **odrębny obiekt** (starsze systemy)

Pojęcie Proces stosuje się wyłącznie do programów użytkownika,

Kod SO przetwarzany jest jako odrębny Obiekt, działający w trybie uprzywilejowanym.

Przetwarzanie **jądra** systemu operacyjnego **przebiega** poza jakimkolwiek procesem.

SO posiada własny obszar pamięci oraz osobny stos systemowy służący do sterowania wywołaniami procedur i poleceniami powrotu.

Kiedy **Proces** zostaje przerwany lub spowoduje wywołanie administracyjne, wówczas jego **kontekst** jest zachowywany, a sterowanie przekazywane do jądra.

SO może po wykonaniu zaleconych funkcji:

-odtworzyć kontekst przerwanego procesu i wznowić jego przetwarzanie,

-zapisać kontekst procesu i uruchomić procedury szeregowania aby wybrać inny proces.

Zachowanie się systemu zależy od przyczyny przerwania i okoliczności.

❒ **System operacyjny, jako Proces Użytkownika** (komputery osobiste, stacje robocze)

System operacyjny jest zbiorem procedur wykonujących na żądanie użytkownika różne funkcje systemowe i działające wewnątrz uruchomionego przez użytkownika Procesu.

System operacyjny może zarządzać w dowolnej chwili pewną liczbą obrazów procesów, których strukturę przedstawia rysunek.

Stos jądra służy do zarządzania wywołaniami i rozkazami powrotu, kiedy proces znajduje się w trybie jądra.

Kod i **dane Systemu Operacyjnego** znajdują się we **wspólnej** przestrzeni adresowej współużytkowanej przez wszystkie procesy użytkownika.

☛ **Wystąpienie przerwania, pułapki** lub wywołania administracyjnego, przełącza CPU w **tryb jądra** i przekazuje sterowanie ***SO***.

Zapisany zostaje kontekst trybu i przełączenie na procedurę SO, której przetwarzanie odbywa się w ramach bieżącego procesu **użytkownika**.

Zmianie ulega tylko **tryb pracy** procesu i **nie** zachodzi przełączenie CPU na inny proces.

Po wykonaniu swoich zadań SO może wznowić bieżący proces poprzez **przełączenie trybu.**

Program użytkownika można przerwać, aby wykonać pewne procedury SO, po czym wznowić go bez ponoszenia kosztów związanych z przełączaniem między procesami.

Jeżeli wystąpi konieczność uruchomienia innego Procesu niż ostatnio przerwanego, to SO uruchamia procedurę **przełączania Procesów**, **która** może być wykonywana w ramach bieżącego Procesu lub nie.

Jednak w pewnej chwili bieżący Proces zostaje unieruchomiony, a sterowanie zostaje przekazane innemu.

Proces może w pewnej chwili zapisać informacje o swoim stanie, wybrać do uruchomienia inny gotowy proces i oddać mu sterowanie.

W krytycznym momencie kod wykonywany w **procesie użytkownika** jest kodem systemu operacyjnego, a **nie** programu użytkownika.

☛ Użytkownik nie może wpływać na działanie procedur SO (chociaż wykonywane są w otoczeniu **jego Procesu**), ze względu na przełączanie trybu użytkownik/jądro.

W trakcie jednego procesu wykonywane są: program użytkownika i programy SO.

❒ **System operacyjny** jako **zbiór Procesów** (środowiska wieloprocesorowe)

Najważniejsze funkcje jądra zorganizowane są jako oddzielne Procesy.

Funkcje jądra wykonywane jako Procesy mogą działać z zadanym priorytetem i być przeplatane innymi Procesami pod kontrolą dyspozytora lub na osobnych procesorach.

Oprócz nich może występować pewna liczba **programów**, **przełączanych** przez **Procesy**, które są wykonywane poza wszelkimi procesami.

☛ Pozwala to implementować niekrytyczne funkcje SO jako odrębne Procesy.

Przykładem może być program monitorujący wykorzystanie zasobów (procesora, pamięci), który nie świadczy konkretnych usług dla aktywnych procesów, może być jedynie wywoływany przez system operacyjny.

Mikrojądro

Mikrojądrem określane jest niewielkie jądro systemu operacyjnego, które jest fundamentem dla modularnych rozszerzeń.

W systemach operacyjnych o strukturze warstwowej funkcje zorganizowane są hierarchicznie, a wzajemne oddziaływania zachodzą wyłącznie między sąsiadującymi warstwami.

Większość warstw działają w trybie jądra.

Każda warstwa posiada znaczną funkcjonalność.

Duże zmiany w jednej warstwie mogą skutkować następstwami, często trudnymi do wyśledzenia, w warstwach sąsiednich.

Modyfikowanie systemu operacyjnego poprzez ujmowanie lub dodawanie funkcji okazuje się praktycznie zadaniem skomplikowanym.

Filozofia mikrojądra zakłada, że należeć do niego mogą tylko **podstawowe** funkcje systemu operacyjnego.

Usługi o mniejszym znaczeniu oraz aplikacje powinny być budowane poza mikrojądrem i działać w trybie **użytkownika**.

Do tej grupy należą między innymi sterowniki urządzeń, systemy plików, menedżer pamięci wirtualnej, system wyświetlania okienek oraz usługi zabezpieczające.

- Składniki systemu operacyjnego zewnętrzne w stosunku do mikrojądra są **implementowane jako procesy usługowe**, które współdziałają między sobą na ogół za pomocą **komunikatów** przekazywanych za pośrednictwem mikrojądra.

➔ Zadaniem mikrojądra jest **obsługa wymiany komunikatów**, polegająca na ich atestowaniu, przenoszeniu między składnikami i przyznawaniu dostępu do sprzętu.

Jeśli aplikacja chce otworzyć plik, musi przesłać odpowiedni komunikat do serwera systemu plików.

Gdy chce utworzyć proces lub wątek, musi wysłać komunikat do serwera procesów.

Każdy z serwerów może wysyłać komunikaty do innych serwerów i wywoływać elementarne funkcje mikrojądra.

Mamy do czynienia z architekturą klient-serwer zaimplementowaną w obrębie jednego komputera.

Procesy nie muszą rozróżniać usług poziomu jądra od usług poziomu użytkownika, gdyż wszystkie one są świadczone na zasadzie przekazywania komunikatów.

- **Architekturę mikrojądra cechuje łatwa rozszerzalność.**

Można sobie na przykład utworzyć kilka systemów plików dla dysku, zaimplementowanych jako procesy poziomu **użytkownika**, a nie jako usługi plikowe dostępne na poziomie jądra.

Rozszerzalność architektury wiąże się z jej elastyczność.
SO można rozbudowywać o nowe mechanizmy, ale można z niego również usuwać mechanizmy istniejące, celem osiągnięcia mniejszej i wydajniejszej implementacji.

- **Małe mikrojądro można poddać rygorystycznym testom.**

Zastosowanie niewielkiej liczby interfejsów programowania aplikacji (API) podnosi prawdopodobieństwo wyprodukowania wysokiej jakości kodu usług systemu operacyjnego działających na **zewnątrz** jądra.

- **Mikrojądro nadaje się do obsługi systemów rozproszonych.**

Komunikat przesyłany przez klienta do procesu serwera musi zawierać identyfikator wywoływanej usługi.

Jeśli wszystkie procesy i usługi w systemie rozproszony mają niepowtarzalne identyfikatory, to na poziomie mikrojądra istnieje pojedynczy obraz całego systemu.

Proces może wysyłać komunikat bez potrzeby interesowania się, która maszyna świadczy odpowiednią usługę.

- **Potencjalną wadą mikrojąder może być ich wydajność.**

Zbudowanie i przesłanie komunikatu poprzez mikrojądro, następnie zaakceptowanie i odkodowanie odpowiedzi trwa dłużej niż zbudowanie pojedynczego wywołania usługi.

Wpływ innych czynników sprawia jednak, że trudno realny ocenić poziom obniżenia wydajności.

Mikrojądro musi zawierać zarówno funkcje odnoszące się bezpośrednio do sprzętu, jak i niezbędne do obsługi serwerów i aplikacji działających w trybie użytkownika.

Funkcje te można podzielić na trzy kategorie odpowiedzialne za:
-zarządzanie pamięcią niskiego poziomu,
-komunikację międzyprocesową (IPC)
-zarządzanie przerwaniami i operacjami wejścia-wyjścia.

❶ Zarządzanie pamięcią niskiego poziomu

Mikrojądro **steruje** sprzętową organizacją przestrzeni adresowej, zapewniając możliwość stosowania zabezpieczeń na poziomie procesów.

Jeśli mikrojądro odpowiada za odwzorowanie stron wirtualnych na fizyczną ramkę strony, wówczas większość mechanizmów zarządzania pamięcią, w tym zabezpieczanie przestrzeni adresowej jednych procesów przed oddziaływaniem innych, algorytm zastępowania stron oraz logiczna organizacja stronicowania może być zaimplementowane na **zewnątrz** jądra.

Przykładowo moduł pamięci wirtualnej, działający poza jądrem, decyduje, kiedy sprowadzić stronę do pamięci, jak również które strony już rezydujące w pamięci, powinny zostać zastąpione.

➔ Mikrojądro odwzorowuje odwołania do stron na fizyczne adresy w pamięci głównej.

❷ Komunikacja między procesami

W SO opartych na mikrojądrze głównym sposobem komunikacji między procesami i wątkami jest wymiana komunikatów.

Komunikat składa się z nagłówka identyfikującego proces nadający i proces odbierający oraz części głównej, zawierającej przekazywane dane, wskaźnik do bloku lub danych i czasami pewne informacje potrzebne do sterowania procesem.

Komunikacja między procesami odbywa się za pośrednictwem portów związanych z procesami.
Port jest kolejką komunikatów przeznaczonych dla danego procesu.

Port zawiera listę możliwości wskazujących, co inne procesy mogą przekazać danemu procesowi.

➔ **Mikrojądro** odpowiada za rozróżnianie portów i pielęgnowanie ich list możliwości.

Proces może przyznać nowy tryb dostępu do siebie poprzez wysłanie do jądra odpowiedniego komunikatu, wskazującego mu nowe możliwości portu.

Przekazywanie komunikatów między procesami o rozłącznych przestrzeniach adresowych obejmuje kopiowanie danych między różnymi obszarami pamięci, a zatem jego wydajność jest ograniczona szybkością pamięci i nie zależy od szybkości procesora.

Wydajniejsza może być komunikacja między procesami, bazującą na schematach współdzielenia pamięci, jedna strona pamięci jest współużytkowana przez kilka procesów.

❸ Operacje Wejścia-Wyjścia i zarządzanie przerwaniami

Technika mikrojądra dopuszcza możliwość obsługi przerwań sprzętowych za pomocą komunikatów i utrzymywania portów wejścia-wyjścia w przestrzeni adresowej.

➔ **Mikrojądro rozpoznaje** przerwania, ale ich nie obsługuje.

Po odnotowaniu przerwania jądro generuje **komunikat** do procesu działającego w trybie użytkownika odpowiedzialnego za jego obsługę.

Przerwania obsługują specjalne procesy działające w trybie użytkownika, natomiast jądro odpowiada jedynie za ich koordynowanie.

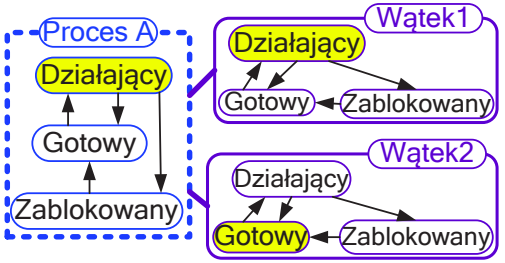
➔ **Mikrojądro przekształca** przerwanie na komunikat, ale nie zajmuje się już jego obsługą na poziomie urządzenia.

ANEKS 4.1. Problem wątków poziomu użytkownika

Jądro zajmuje się szeregowaniem **Procesu** jako całości i ustawia go w określony stan działania (Działający, Gotowy, Zablokowany, itp.).

Jakie relacje zachodzą pomiędzy szeregowaniem **wątków** a szeregowaniem **Procesów**?

Niech **Proces A** zawiera 2 wątki poziomu użytkownika.

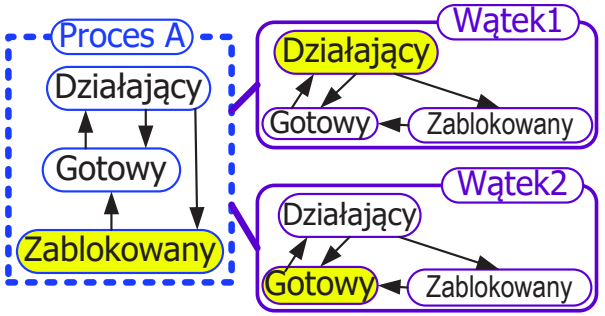


Rys 4.14a. Stany Procesów i Wątków

Mogą wystąpić zdarzenia:

1. W aplikacji działa ***Wątek 1***, który generuje wywołanie systemowe, żądając wykonania operacji We/Wy co blokuje **Proces A**.

Sterowanie przejmuje **jądro**, rozpoczynając operację We/Wy i przenosi **Proces A** w stan zablokowany, uruchamiając inny proces, Rys. 4.14b.



Rys. 4.14b . Stany Procesów i Wątków

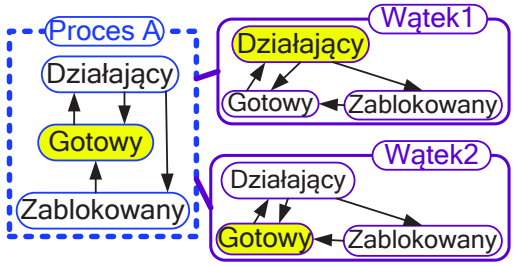
Wątek 1 Procesu **A** pozostaje w stanie Działania.

Wątek 1 Procesu **A** **nie jest fizycznie** podłączony do CPU lecz jest ***postrzegany*** jako ***działający*** przez **bibliotekę** wątków `(ale nie jako przetwarzany)`.

2. Przerwanie zegarowe przekazuje sterowanie do **jądra**, które stwierdza, że wykonywany **Proces A** wyczerpał limit czasu.

Jądro przenosi **Proces A** do stanu **Gotowości** i uruchamia inny proces.

W tym samym czasie ***Wątek 1*** **Procesu A** jest ***postrzegany*** jako działający.



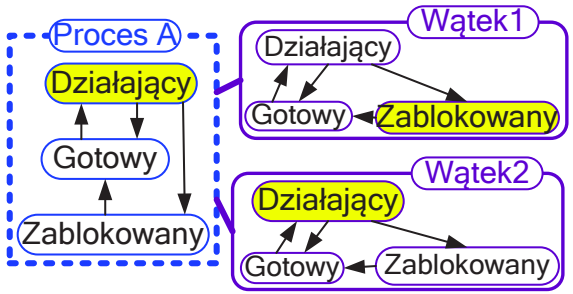
Rys 4.14c. Stany Procesów i Wątków

3. **Proces A** przechodzi w stan Działania.

Wątek 1 przeszedł do linii kodu, w którym potrzebuje, by Wątek 2 **Procesu A** wykonał pewne działania.

Wątek 1 wchodzi w stan zablokowany, a

Wątek 2 ze stanu Gotowości do stanu Działania.



Rys 4.14d. Stany Procesów i Wątków

Dla (1) i (2) kiedy **jądro** przełączy sterowanie z powrotem do **Procesu A**, wykonywanie podejmie ***Wątek 1***.

ANEKS 4.2 Procesy w systemie Unix

Rys. 4.4. Stany procesu UNIX

Dwa stany „**AKTYWNY**", w Unix'ie pozwalają rozróżnić kiedy proces jest wykonywany w trybie użytkownika, a kiedy w trybie jądra.

Zombie –proces, który przestał istnieć, ale pozostawił informacje o swoim procesie rodzicielskim.

Uśpiony (Zablokowany) – proces znajduje się w PAO lecz nie może być uruchomiony zanim nie zajdzie określone **zdarzenie**.

Wywłaszczony: -proces działa w trybie jądra (wywołanie administracyjne, przerwanie zegarowe lub We/Wy); jądro kończy pracę i może przekazać sterowanie programowi użytkownika.

Jądro może zdecydować o wywłaszczeniu bieżącego procesu na korzyść innego ze stanu **Gotowy** i o wyższym priorytecie.

W takim przypadku proces przechodzi do stanu **Wywłaszczony**.

Procesy w stanie **Gotowy** i **Wywłaszczony** są równoprawnie dla mechanizmu szeregowania i oczekują na uruchomienie w tej samej kolejce.

Uwaga: **Wywłaszczenie** może zajść, kiedy proces przechodzi z trybu jądra do trybu użytkownika.

Proces działający w trybie jądra nie mógł zostać wywłaszczony (może od v.2.6).

Funkcja systemowa ***fork*** tworzy nowy proces, któremu nadaje stan **Utworzony**.

Po zakończy procesu tworzenia, jego stan zmienia się na **Gotowy** i oczekuje na przeniesienie do **Aktywnego**.

Po wybraniu procesu do wykonania, rozpoczyna się przełączanie kontekstu, poprzez wywołanie funkcji jądra ***switch()***, która inicjuje rejestry i przekazuje procesowi sterowanie.

Proces wykonujący się w trybie użytkownika wchodzi do trybu jądra gdy:

-nastąpi wywołanie funkcji systemowej,

-pojawi się przerwanie.

Powraca do trybu użytkownika, gdy zakończy się obsługa tych zdarzeń.

Wykonując funkcję systemową proces czasami musi czekać na zajście określonego zdarzenia lub na zwolnienie zajętego w danej chwili zasobu.

Oczekiwanie realizuje wywołanie funkcji ***sleep(),*** która zmienia stan na **Uśpiony**.

Gdy zajdzie oczekiwane zdarzenie lub zwolni się zasób, jądro budzi proces, który przechodzi do stanu **Gotowy**, i czeka na przeniesie do stanu **Aktywny**.

Wybrany proces do wykonania początkowo wykonuje się w trybie jądra **Aktywny_w_trybie_jądra**, w którym kończy czynności związane z przełączaniem kontekstu.

Kolejna zmiana stanu zależy od tego, co proces robił, gdy był wykonywany poprzednio.

Proces nowo utworzony lub wykonujący uprzednio kod użytkownika (i został wywłaszczony, aby mógł się wykonać proces o wyższym priorytecie), powraca natychmiast do trybu użytkownika.

Proces uprzednio wstrzymany w oczekiwaniu na zasób w trakcie wykonywania funkcji systemowej, wznawia wykonanie przerwanej funkcji systemowej w trybie jądra.

Proces kończy swoje działanie: -wywołując funkcję systemową ***exit***;
-na skutek zawiadomienia wysyłanego przez jądro (***sygnał***).

W obu wypadkach jądro zwalnia zasoby kończącego się procesu, z wyjątkiem informacji o:
-***kodzie zakończenia***,
-***wykorzystaniu zasobów***, i zmienia stan procesu na **Zombie**.

Proces pozostaje w stanie **Zombie**, dopóki jego proces macierzysty nie wywoła funkcji ***wait***, która usunie proces z systemu i przekaże jego kod zakończenia do procesu macierzystego.

W niektórych systemach proces może być **Zatrzymany** lub **Zawieszony** przez sygnał ***stop*** .

Sygnał ***stop*** zmienia natychmiast stan procesu:
-jeśli jest w stanie **Wykonywany** lub **Gotowy**, to jego stan zmienia się na **Zatrzymany**.
-jeśli jest w stanie **Uśpiony**, to wchodzi w stan **Zatrzymany i Uśpiony**.

Proces zatrzymany może być wznowiony przez sygnał ***wznowienie*** (SIGCONT), który przywraca go do stanu **Gotowy**.

Jeśli proces był jednocześnie w stanie **Zatrzymany** i **Uśpiony**, to sygnał SIGCONT spowoduje zmianę jego stanu na **Uśpiony**.

W Uniksie występują dwa specjalne procesy: ***proces_0***, tworzony w chwili uruchamiania systemu oraz generowany przezeń ***proces_1***, zwany inicjującym. Pozostałe procesy w systemie są procesami potomnymi ***procesu_1***.

ANEKS 4.3. Procesy i Wątki w systemie WINDOWS

Jądro Windows jest obiektowe, zaś jego strony nie są usuwane z PAO, nie podlega skutkom wywłaszczania.

Obiektowość jądra rozumiana jest jako **typ** danych, mający zbiór atrybutów i metod.

Jądro wykonuje swoje zadania, posługując się zbiorem ObiektówJądra, których **atrybuty** reprezentują dane jądra, a **metody** realizują działania jądra.

A 4.3.1. Procesy

Proces jest `egzemplarzem` działającego **programu**, w skład którego wchodzą:

1. **ObiektJądra**, za pomocą którego system zarządza procesem i przechowuje statystyczne informacje o procesie.
2. **Przestrzeń adresowa** zawierająca kod i dane modułu wykonywalnego lub DLL, oraz pamięć alokowana dynamicznie na stosy lub sterty wątku.

Proces jest bezczynny i **nigdy nie otrzymuje czas CPU**.

Musi uruchomić przynajmniej jeden **wątek** w swoim kontekście, który wykona kod zawarty w przestrzeni adresowej **procesu**.

Uruchamiając proces system automatycznie tworzy jego pierwszy wątek, zwany głównym, który może tworzyć wątki dodatkowe (potomne).

Gdyby zabraknie wątku wykonującego kod w przestrzeni adresowej procesu, system automatycznie usuwa proces z jego przestrzenią adresową.

Funkcja *CreateProcess* jest funkcją systemu Windows, tworzącą nowy proces.

Po wywołaniu **CreateProcess**, system operacyjny tworzy:

- w jądrze **ObiektProces** z licznikiem użyć ustawionym na 1; jest to strukturą, której system używa do zarządzania procesem, przechowywania informacji statystycznych o procesie.
- w jądrze **ObiektWątek** (licznik użyć=1), odpowiadający głównemu wątkowi nowego procesu; jest to strukturą danych używaną przez SO do zarządzania właściwym wątkiem.
- **wirtualną przestrzeń adresową** nowego procesu i ładuje do niej kod oraz dane z pliku wykonywalnego, a także potrzebne moduły DLL.

Następnie funkcja **CreateProcess** otwiera **O**biekt**P**roces oraz **O**biekt**W**ątek i umieszcza ich uchwyty (względem procesu) w składowych *hProcess* i *hThread parametru* ***lpProcInfo***.

Po otwarciu tych Obiektów ich liczniki użyć przyjmują wartość **2**.

A4.3. 2. Wątki

Funkcja CreateThread jest **funkcją systemu Windows, tworzącą nowy wątek.**

Jeżeli utworzenie wątku **powiedzie się** zwraca jego **uchwyt**.

Wywołanie funkcji **CreateProcess** tworzy **wątek podstawowy** (*primary thread).*

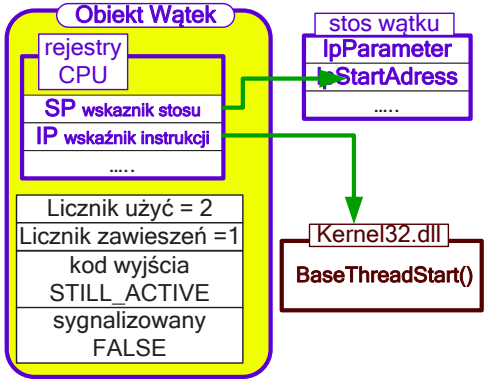
- ☛ Utworzenie pochodnych wątków wymaga wywołania funkcji **CreateThread** w już działającym wątku.

Wywołanie funkcji **CreateThread:**

-utworzy w jądrze **ObiektWątek** (który pozostaje w systemie, dopóki nie zostanie zamknięty jego uchwyt),

-alokuje pamięć przeznaczoną na stos wątku (pamięć z przestrzeni adresowej procesu, wątek nie dysponuje własną przestrzenią).

Uwaga: Stosy wątków rosną od wyższego adresu pamięci do niższego.



Rys. 4.16. Struktura wątku w WINDOWS

Inicjalizacja parametrów **ObiektWątek**:

-licznik użyć = 2,

-licznik zawieszeń = 1,

-kod wyjścia = STILL_ACTIVE (0x103),

-stan obiektu = niesygnalizowany.

System zapisuje w stosie nowego wątku wartości następujących parametrów funkcji **CreateThread**:

-lpParameter,

-lpStartAddress.

Wątek działa zawsze w kontekście procesu.

Każdy wątek ma własny zbiór rejestrów CPU zwany ***kontekstem*** wątku, który odzwierciedla stan rejestrów CPU podczas ostatniego wykonywania wątku.


```
    struct PARM {
            int nData;   double a, b, *X;  char zn;
            };
int main()                                          // Przyklad_Watki
{
int nData = 10000;
double  *A = new double[nData], *B = new double[nData];
   PARM   par1 = { nData,   1.1,   9.9,     A,  'A' },
          par2 = { nData,  10.1, 19.9,     B,  'B' };
 CreateThread(NULL,  0, Gen,   &par1,  0,  &ID1);
 CreateThread(NULL,  0, Gen,   &par2,  0,  &ID2);
 CreateThread(NULL,  0, Sort,  &par1,  0,  &ID3);
 CreateThread(NULL,  0, Sort,  &par2,  0,  &ID4);
          WatekGlowny(100, '*');
cout << "Koniec";
delete [ ] A, B;
return 0;
}
```

➔ Pisząc wielowątkowe programy w C++ zaleca się używać funkcji biblioteki run-time: **_beginthread** i **_beginthreadex**. ich deklaracje zawiera plik <**process.h**>.

Pozwala to chronić wielowątkową aplikację przed niesynchronizowanym dostępem do danych w C++.

Windows obsługuje sprzętową konfigurację **SMP** (Symetryczne Przetwarzanie Wieloprocesorowe)

Wątki każdego procesu mogą być przetwarzane na dowolnym CPU.

- ☛ Wątki tego samego procesu mogą być równolegle przetwarzane na różnych CPU.

Każdy wątek ma przypisany priorytet, będący liczbą z przedziału od **0** (najniższy) do **31** (najwyższy).

Każdy proces ma przypisaną **pewną klasę** priorytetu.

Wątki mają priorytety ustawione względem procesu, który je stworzył.

Jakie numery z zakresu (0 ÷ 31) otrzymują wątki?

System sam mapuje klasy priorytetów procesów i względne priorytety wątków na wartości numeryczne.

Przykładowe atrybuty procesów w systemie Windows

Identyfikator procesu	Identyfikuje proces w systemie.
Deskryptor zabezpieczeń	Ustala twórcę obiektu, kto ma do niego dostęp, kto może go wykorzystywać.
Priorytet podstawowy	Główny priorytet dla celów szeregowania **wątków** procesu.
Rodzina CPU	Grupa CPU na których mogą być przetwarzane **wątki**.
Przydzielone limity	Ilości pamięci systemowej, stronicowanej przestrzeni plików oraz czasu CPU.
Czas przetwarzania	Całkowity czas, w jakim mogą być przetworzone wszystkie wątki procesu.

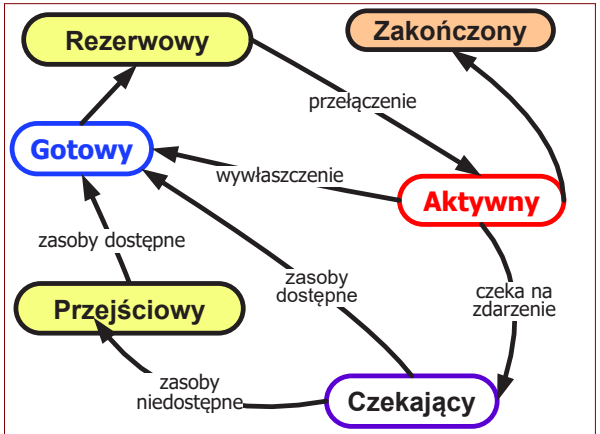
❒ Stany wątku w systemie Windows

Gotowy: zdolny do natychmiastowego działania, pod kontrolą dyspozytora, który szereguje je stosownie do posiadanych priorytetów.

Aktywny: przetwarza do czasu wywłaszczenia, wyczerpania limitu czasu lub zakończenia.

Oczekujący: przyczyny czekania -czeka na zdarzenie (np. zakończenia operacji We/WY),
-przestój dla potrzeb synchronizacji,
-zlecenia podsystemu środowiskowego.

Po wystąpieniu oczekiwanych okoliczności wątek przechodzi w stan **Gotowy**.



Rys. 4.17. Stany wątku w Windows

Zakończony: może zostać zakończony samoistnie, przez inny wątek lub z powodu zakończenia procesu rodzicielskiego.
Wątek zostaje usunięty z systemu lub zachowany celem ponownej inicjacji w przyszłości.

Przejściowy: jest gotowy do działania, ale nie są dostępne wszystkie jego zasoby, (np.: przeniesienia jego stosu do obszary wymiany).
Gdy zasoby wątku staną się dostępne, przejdzie w stan **Gotowy**.

Rezerwowy (Standby): oczekuje na uruchomienie na danym CPU natychmiast, gdy stanie się on dostępny.
Jeśli priorytet wątku zapasowego jest wystarczająco wysoki, wówczas wątek aktualnie działający na danym CPU może zostać wywłaszczony na jego korzyść.
W przeciwnym razie wątek zapasowy oczekuje, aż wątek bieżący wyczerpie swój limit czasu CPU.

Anex 4.4. Wątki poziomu użytkownika ULT a wątki poziomu jądra KLT

Poniższa tabela zawiera wyniki pomiarów wykonanych na jednoprocesorowej maszynie VAX, działającej pod kontrolą systemu operacyjnego klasy Unix.

Czasy oczekiwania na wykonanie operacji przez wątki i procesy.

Operacja	Wątki poziomu użytkownika	Wątki poziomu jądra	Procesy
Null Fork	34	948	11300
Signal Wait	37	441	1840

Null Fork — czas mierzony od utworzenia (poprzez szeregowanie, wykonywanie), do zakończenia procesu (wątku), który wywołuje pustą procedurę czyli obciążenie powodowane przez utworzenie procesu (wątku).

Signal Wait— czas potrzebny, aby proces (wątek) przesłał sygnał oczekiwania innemu procesowi (wątkowi) i otrzymał odpowiedź zawierającą warunek, czyli obciążenie powodowane wzajemną synchronizacją procesów (wątków).

❒ Rozwiązania mieszane

Niektóre operacyjne obsługują mechanizmy mieszane oparte na obu typach wątków (np. Solaris).

W systemach mieszanych tworzenie wątków odbywa się w przestrzeni użytkownika dla większości wątków szeregowanych i synchronizowanych w aplikacji.

- Wiele wątków poziomu użytkownika z pojedynczego programu użytkownika zostaje odwzorowanych na większą lub mniejszą liczbę wątków poziomu jądra.
- Programista może dostosować liczbę wątków poziomu jądra dla konkretnej aplikacji i dla konkretnej maszyny tak, by osiągnąć najlepszy wynik.
- W rozwiązaniach mieszanych liczne wątki tej samej aplikacji mogą działać równolegle na kilku procesorach, a blokujące wywołania systemowe nie muszą blokować całego procesu.

Dobrze zaprojektowany system mieszany powinien zagwarantować większość korzyści stosowania wątków **KLT** i **ULT** w czystej postaci, minimalizując ich wady.

W tradycyjnym ujęciu, szeregowaniu podlega obiekt zrealizowany jako pojedynczy Proces.

Takie podejście wyraża relację między wątkami a procesami w stosunku **1:1**.

Obecnie rozwiązania stosują wielu wątków w ramach jednego procesu, relacja wiele do jednego.

Liczba wątków do procesów	Opis	Przykłady systemów
1:1	Każdy wątek realizuje pojedynczy proces, posiadający własną przestrzeń adresową i zasoby.	Tradycyjne implementacje Unix
M:l	Proces definiuje przestrzeń adresową i dynamiczne własność zasobów. W ramach jednego procesu może być tworzonych wiele wątków.	Windows, Solaris, Linux, OS/2
1:M	Wątek może migrować między środowiskami różnych procesów. Można przenosić wątek pomiędzy różnymi systemami.	Clouds, Emerald
M:N	Kombinacja cech rozwiązań M:1 i l:M.	TRIX

❒ **Relacja jeden** do **wielu**

W rozproszonych systemach operacyjnych modna jest idea wątku traktowanego jako obiekt.

Cechą tego rozwiązania jest możliwości przenoszenia wątku między przestrzeniami adresowymi.

W systemie operacyjny Clouds, wątek jest jednostką działania widzianą z perspektywy użytkownika.

Proces składa się z wirtualnej przestrzeni adresowej oraz związanego z nią bloku sterowania procesem. Po utworzeniu wątek rozpoczyna działanie, przywołując w ramach procesu punkt wejściowy do programu.

Wątek może przenosić się między przestrzeniami adresowymi, jak też przechodzić z jednego komputera na inny.

Podczas przenoszenia wątek musi dbać o zachowanie różnych informacji, takich jak dane o terminalach sterujących, parametry ogólne oraz zalecenia dotyczące szeregowania (priorytety).

Mechanizmy systemu Clouds skutecznie izolują użytkowników i programistów od szczegółów rozproszonego środowiska.

Działalność użytkownika może być reprezentowana jako jeden wątek, którego przenoszenie miedzy różnymi maszynami powierza się systemowi operacyjnemu.

Takie podejście opłaca się na w przypadku konieczności zapewnienia dostępu do zdalnych zasobów oraz równomiernego obciążenia.

❒ **Relacja wiele** do **wielu**

Idea ta wykorzystuje pojęcie domena i wątku.

Domena jest jednostką statyczną, wyposażoną w przestrzeń adresową i ***porty***, przez które mogą być wysyłane i przyjmowane komunikaty.

Wątek to pojedyncza ścieżka przetwarzania, posiadająca stos wykonawczy i kontekst przechowujący stan procesora oraz informacje dotyczące szeregowania.

Kilka wątków może działać w ramach jednej domeny.

Działanie pojedynczego użytkownika lub aplikacji można realizować w kilku domenach.

W takim przypadku istnieje wątek, który może się przenosić między domenami.

● Rozważmy program korzystający z podprogramu obsługi operacji We/Wy.

W środowiskach wielowątkowych, Program Główny może wygenerować nowy proces do obsługi operacji We/Wy i kontynuować dalsze działanie.

Jeżeli dalszy postęp realizacji Programu Głównego zależy od wyniku operacji We/Wy, będzie on musiał czekać na jej zakończenie.

Powyższy problem można implementować na kilka sposobów.

❶ Cały program można zaimplementować jako pojedynczy proces.

Wydajne przetwarzanie całego procesu może wymagać zaangażowania dużej ilości pamięci.

Podprogram obsługujący operacje We/Wy wystarczy niewielki bufor i *trochę* pamięci na kod.

Podprogram obsługi We/Wy działa w przestrzeni adresowej dużego programu, zatem:

-cały proces musi pozostawać w pamięci głównej na czas przetwarzania operacji We/Wy, lub

-cały proces będzie przeniesiony do obszaru wymiany.

Ten sam efekt osiąga się implementując podprogram obsługi We/Wy w postaci dwóch wątków w tej samej przestrzeni adresowej.

❷ Program główny oraz podprogram We/Wy implementuje się jako dwa oddzielne procesy.

Takie rozwiązanie obciąża SO obowiązkiem stworzenie procesu podrzędnego.

Jeśli operacje We/Wy będą się często powtarzać, proces podrzędny musi być:

-utrzymywany przy życiu, co wymaga **blokady** pewnej ilości pamięci, albo

-często zatrzymywany i ponownie aktywowany, co **obniży** wydajność.

❺ Program Główny oraz podprogram We/Wy traktuje się jako **pojedyncze** działanie.

Jest ono zaimplementowane jako **pojedynczy** wątek,

ale tworzy się dla nich **odrębne** **przestrzenie** adresowe (domeny).

Wątek może być w trakcie wykonywania programu przenoszony między obiema przestrzeniami adresowymi.

System operacyjny może nimi administrować niezależnie i nie występuje dodatkowe obciążenie związane z tworzeniem procesu podrzędnego.

Dodatkowy zysk to możliwość współdzielenia przestrzeni adresowej podprogramu obsługi We/Wy z innymi prostymi programami o podobnym przeznaczeniu.

Symetryczne wieloprzetwarzanie (*Symmetric Multi Processing* **SMP**)

Co około **20** ms Windows przegląda wszystkie istniejące w **jądrze ObiektyWątki**.

Niektóre z nich nadają się do wykonania.

System wybiera jeden z tych Obiektów i ładuje zawartość jego kontekstu do rejestrów CPU - ***przełączenia kontekstu***.

Wątek uzyskuje dostęp do CPU i wznawia wykonanie kodu, realizując operacje na danych w przestrzeni adresowej procesu.

Gdy minie dalsze **20** ms, Windows zapisuje rejestry CPU z powrotem w kontekście wątku i przerywa jego działanie.

System ponownie wybiera jeden ObiektWątek z puli Obiektów gotowych do wykonania, ładuje kontekst do rejestrów CPU i wznawia jego działanie.

Windows rejestruje liczbę przełączeń kontekstu każdego wątku.

Nie można zagwarantować uruchomienie wątku w **określonym czasie** od **pewnego zdarzenia.**

System przydziela **CPU** tylko wątkom, które są zaszeregowane do wykonania.

Niektóre ObiektyWątki mogą mieć licznik zawieszeń ustawiony na wartość większą od zera.

Oznacza to, że odpowiadające im wątki są Zawieszone.

A4.5. Włókno *(fiber)*

Serwerowe aplikacje systemy UNIX są jednowątkowe (z punktu widzenia Windows), lecz mogą obsługiwać wielu klientów.

Aby ułatwić konwersję wyposażono WINDOWS w mechanizm włókna.

Włókno to kod działający w trybie **użytkownika**, dla którego przydział CPU jest w pełni sterowany przez użytkownika.

Proces może mieć **wiele** włókien, lecz nawet w środowisku wieloprocesorowym może działać **tylko jedno** włókno w danym czasie (wątki mogą działać współbieżnie).

* Włókno nie podlega wywłaszczaniu z punktu widzenia jądra.

➔ Na początku **należy zamienić** istniejący wątek we włókno (CorvertThreadToFiber())

LPVOID ConvertThreadToFiber(LPVOID lpParameter // fiber data for new fiber);

Funkcja zwraca adres utworzonego włókna lub NULL (więcej w GetLastError).

Funkcja alokuje pamięć dla kontekstu wykonania włókna, i następuje kojarzy adresu tego kontekstu z wątkiem.

Wątek stał się włóknem i teraz to włókno działa w imieniu wątku.

Jeśli włókno (wątek) zakończy działanie lub wywoła *ExitThread,* zniknie włókno i wątek.

Nie warto konwertować wątku na włókno, jeśli nie będą uruchamiane inne włókna, działające w ramach tego samego wątku.

➔ Następne włókna można utworzyć **wyłącznie** w działającym aktualnie włóknie.

Przekonwertowany wątek, poprzez wywołanie funkcji **CreateFiber.**

```
LPVOID CreateFiber(
  DWORD                 dwStackSize,     // initial thread stack size, in bytes
  LPFIBER_START_ROUTINE lpStartAddress,  // pointer to fiber function
  LPVOID                lpParameter      // argument for new fiber
);
```

Funkcja zwraca adres utworzonego włókna lub NULL (więcej w GetLastError).

Argument ***lpStartAddress*** podaje adres bazowej funkcji włókna (wątka) o prototypie:

VOID WINAPI **FiberFunc**(LPVOID ***lpParameter***);

Przy wykonaniu włókna, funkcja włókna uruchamiana jest z parametrem ***lpParameter***.

Po powrocie z funkcji włókna wątek i wszystkie utworzone w nim włókna zostają natychmiast usunięte.

➔ **CreateFiber nie rozpoczyna** wykonywania włókna ⇐ robi to funkcja **SwitchToFiber**.

VOID **SwitchToFiber(** LPVOID lpFiber // pointer to fiber to switch to **)**;

Tylko wywołanie funkcji ***SwitchToFiber*** przydziela CPU dla włókna.

Kod użytkownika jawnie wywołuje tę funkcję ⇐sprawuje kontrolę nad szeregowaniem włókien.

Wątek, w którym działa włókno, może zostać wywłaszczony przez SO.

Podczas wykonywania wątku działa w nim tylko **jedno** włókno wybrane jawnie przez ***SwitchToFiber***.

Aplikacja kończy włókno, wywołując:

VOID **DeleteFiber**(LPVOID lpFiber // pointer to the fiber to delete);

Funkcja ***DeleteFiber*** wywoływana przez jedno włókno usuwa drugie włókno.

Jeśli ***DeleteFiber*** dostanie adres włókna skojarzonego aktualnie z wątkiem, to wywoła w środku *ExitThread,* i zakończy ten wątek oraz wszystkie utworzone w nim włókna.

Wątek może wykonywać jedno włókno jednocześnie i SO zawsze wie, które włókno jest aktualnie skojarzone z wątkiem.

Funkcja ***GetCurrentFiber*** zwraca adres aktualnie działającego włókna.

```
int WINAPI WinMain(HINSTANCE h1, HINSTANCE hPrev1, LPSTR Cmd, int uC)
{
 PVOID pFiberConv = NULL, pFiber1 = NULL;
pFiberConv  =  ConvertThreadToFiber(NULL);
        ...
pFiber1 = CreateFiber(0, MojaFun,  &parametr);
        ...
        ...
SwitchToFiber(pFiber1);
 return 0;
}

void WINAPI MojaFun(LPVOID parametr)
{
        Ciało funkcji
}
```