Systemy Operacyjne

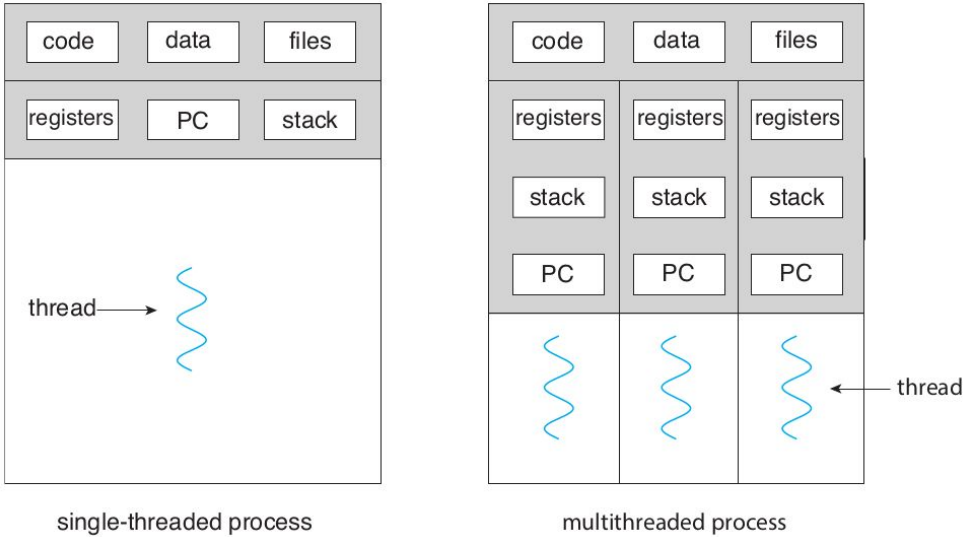
Wątki

Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*

Wątek

- Wątek to podstawowa jednostka wykorzystania procesora.
- Wątek zawiera: identyfikator (numer), licznik programu, rejestry oraz stos.
- Wątek współdzieli z innymi wątkami należącymi do tego samego procesu: sekcję kodu, sekcję danych oraz inne zasoby systemu operacyjnego (np. otwarte pliki, sygnały).



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Wątek

Proces

- Proces osadzony jest w dwóch charakterystykach:
  - Właściciela zasobów, w tym przestrzeni adresowej zawierającej obraz procesu.
  - Planowania i wykonywania, która przeplata się razem z innymi procesami.
- Te dwie wyżej wymienione charakterystyki są niezależnie traktowane w systemie operacyjnym.

Wątek

- Jednostką wykonywanego zadania jest *wątek*, inaczej lekki proces (ang. *lightweight process*).
- Właściciel określony jest przez proces (zadanie), do którego wątek należy.

Proces

- Proces to wykonywany program.
- *Heavy weight process.*
- Czaso- i zasobochłonne: tworzenie, terminacja, przełączanie kontekstu.
- Komunikacja: pamięć dzielona lub wymiana komunikatów jako mechanizmy specjalne.
- Procesy są izolowane.
- Przełączanie procesów odbywa się przez funkcje systemowe.
- Dla jądra dwa procesy to dwa procesy.
- Zablokowanie jednego procesu nie wpływa na fakt zablokowania innego procesu.

vs. Wątek

- Wątek to część danego procesu.
- *Lightweight process.*
- Szybsze i zużywające mniej zasobów na tworzenie, terminację i przełączanie kontekstu.
- Komunikacja: bezpośrednio współdzielone wszystkie zasoby danego procesu.
- Wątki współdzielą.
- Przełączanie wątków odbywa się bez wywoływania przerwań do jądra.
- Dla jądra dwa wątki to jeden proces.
- Zablokowanie procesu, to zablokowanie jego wszystkich wątków.

Proces

- Zablokowanie się procesu macierzystego uniemożliwia tworzenie procesów potomnych.
- Proces ma własny PCB, stos oraz przestrzeń adresową.
- Zmiany w procesie macierzystym nie mają wpływu na procesy potomne.

vs. Wątek

- Zablokowanie pierwszego wątku nie wpływa na działanie pozostałych wątków procesu.
- Ma rodzica PCB, własny TCB, stos oraz współdzieloną przestrzeń adresową.
- Zmiany w procesie wpływają na zmiany w wątkach tego procesu.

Obserwowanie wątków w systemie Linux

```
$ ps -T -o pid,tid,comm -p `pidof insync`
    PID     TID COMMAND
   4224    4224 insync
   4224    4260 QXcbEventQueue
   4224    4280 QDBusConnection
   4224    4281 gmain
   4224    4282 gdbus
   4224    4284 insync
   4224    4312 insync
```

Proces

Wątki

Nazwa wątku, nazwa procesu

```
$ pstree -p `pidof insync`
insync(4224)-+-QtWebEngineProc(4336)---QtWebEngineProc(4338)---QtWebEngineProc(4357)-+-{QtWebEngineProc}(4358)
             |                                                                        |-{QtWebEngineProc}(4360)
             |                                                                        |-{QtWebEngineProc}(4361)
             ...
             |-{insync}(4260)
             |-{insync}(4280)
             |-{insync}(4281)
```

Proces

Wątki

Procesy potomne

PID vs. TID

- PID = *process identifier*
- TID = *thread identifier*
- Jeśli proces ma tylko jeden wątek, to PID == TID
- Jeśli proces ma wiele wątków, to pierwszy z nich ma unikalny TID w zakresie tego procesu
- Jądro systemu nie rozróżnia szczególnie *wątku* od *procesu*
- Dla jądra wątki to procesy, które współdzielą pewne zasoby
- Kiedy tworzymy nowy proces za pomocą `fork()`, to otrzymuje on nowy PID i TID (PID==TID)
- Kiedy tworzymy nowy wątek to otrzymuje on PID taki jak procesu oraz nowy TID.
- Alias: LWP = *Light-Weight Process*

Obserwowanie wątków kernela w Linux

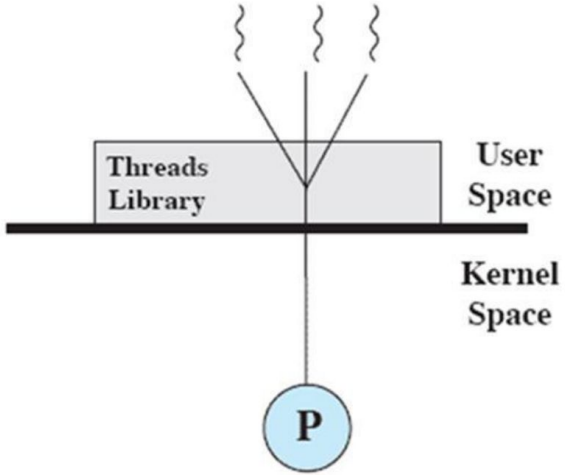
```
$ pstree -p 2
kthreadd(2)-+-UVM Tools Event(977)
            |-UVM deferred re(976)
            |-UVM global queu(975)
            |-acpi_thermal_pm(134)
            |-ata_sff(117)
            |-blkcg_punt_bio(115)
            |-charger_manager(164)
            |-cpuhp/0(14)
```

Dlaczego mają odmienny PID ?

Dlatego, że to jest TID.

Wątki na poziomie użytkownika

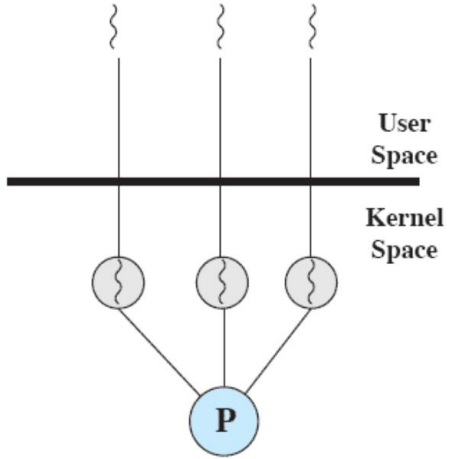
- Zarządzanie wątkami odbywa się na poziomie aplikacji.
- Jądro nie ma wiedzy na temat wątków.



Źródło: TODO

Wątki na poziomie jądra

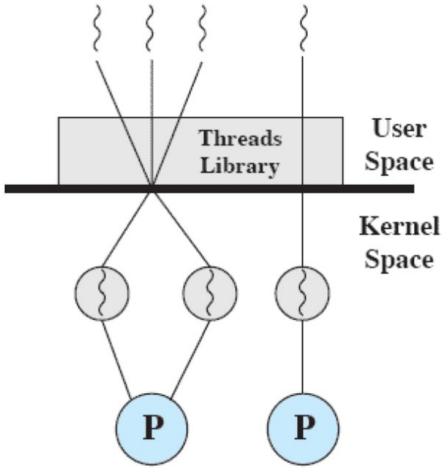
- Jądro zarządza kontekstem dla procesu oraz wątków. Nie ma zarządzania wątkami na poziomie aplikacji.
- Zalety:
  - Kernel może jednocześnie planować realizację wielu wątków z jednego procesu na wielu procesorach/rdzeniach.
  - Jeśli jeden wątek w procesie jest zablokowany, jądro może planować inny wątek tego samego procesu.
  - Funkcjonalność jądra może być wielowątkowa.
- Wada: przekazanie kontroli między wątkami w obrębie procesu wymaga kernel-mode.



Źródło: TODO

Rozwiązanie łączone

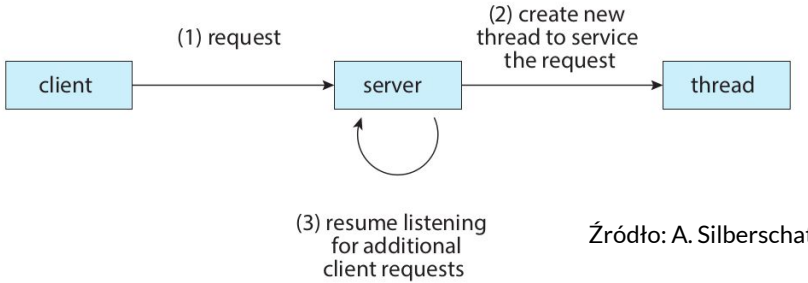
- Tworzenie wątku odbywa się w przestrzeni użytkownika.
- Planowanie (*scheduling*) oraz synchornizacja wątków odbywa się w jądrze



Źródło: TODO

Wątki - zastosowania

- Program serwera obsługujący żądania (ang. *requests*) programów klienckich:
  - Serwer stron WWW i przeglądarka internetowa.
  - Serwer poczty elektronicznej (ang. *mail transfer agent*) i program pocztowy.
  - Sprawdzanie pisowni w edytorze tekstu.
- Prowadzenie obliczeń macierzowych:
  - Wykonywanie operacji na tych samych danych.

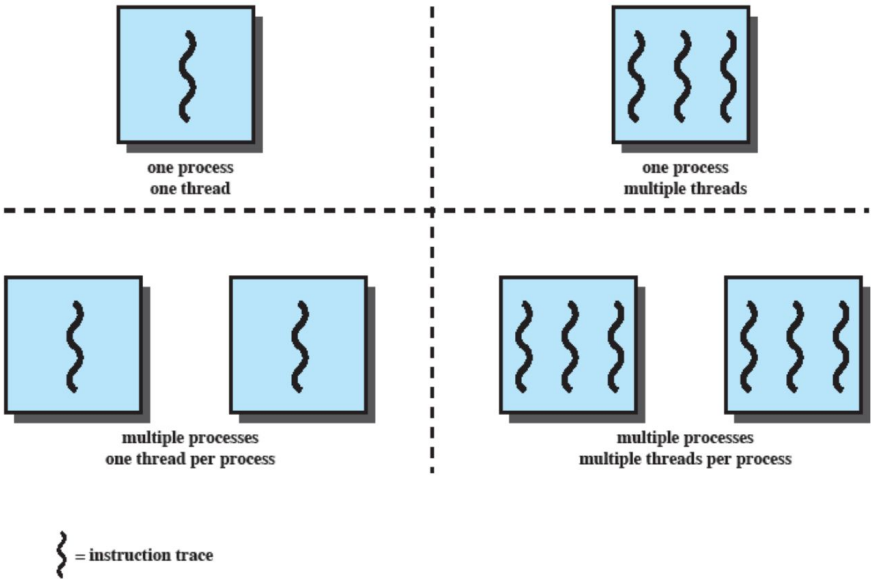


Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Uwaga! Tworzenie wątku jest mniej obciążające niż tworzenie nowego procesu.

Wielowątkowość

- Zdolność systemu operacyjnego do wspierania wielu ścieżek wykonywania w obrębie jednego procesu.
- MS-DOS - single user process with single thread.
- Some UNIX - multiple user processes with single thread per process.
- Java run-time env. - single process with multiple threads.
- Windows, Solaris, modern UNIX, Linux, etc. - multiple processes with multiple threads per process.

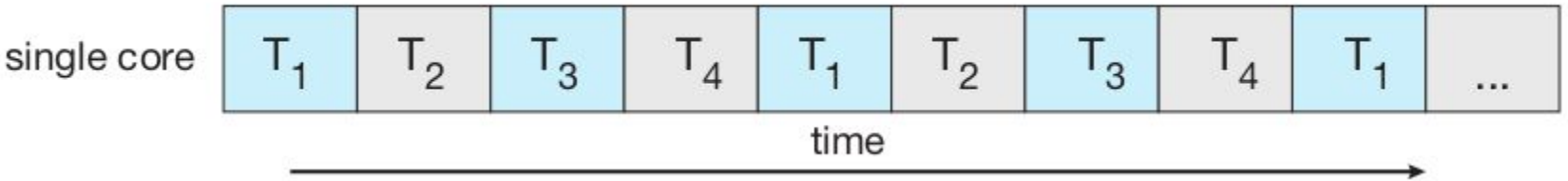


Źródło: TODO

Zalety oprogramowania wielowątkowego

- **Responsywność** - jeśli część aplikacji jest zablokowana, inna jej część może wykonywać operacje, a cała aplikacja sprawia wrażenie ciągłego działania. Zastosowanie: w aplikacji, kiedy jedna wywołana operacja wykonywana jest w tle, interfejs użytkownika pozostaje responsywny.
- **Współdzielenie zasobów** - w przypadku procesów współdzielenie zasobów odbywa się tylko poprzez pamięć współdzieloną, albo przesyłanie komunikatów. Wątki współdzielą zasoby wprost. Współdzielenie kodu i danych umożliwia wątkom działać w tej samej przestrzeni adresowej.
- **Ekonomia** - alokowanie pamięci i zasobów przy tworzeniu procesu jest bardziej kosztowne, niż w przypadku wątków. Przełączanie przełączanie kontekstu jest także szybsze w przypadku wątków.
- **Skalowalność** - aplikacje wielowątkowe mogą działać w architekturze wielordzeniowej. Aplikacja jednowątkowa może być wykonana tylko na jednym rdzeniu procesora.

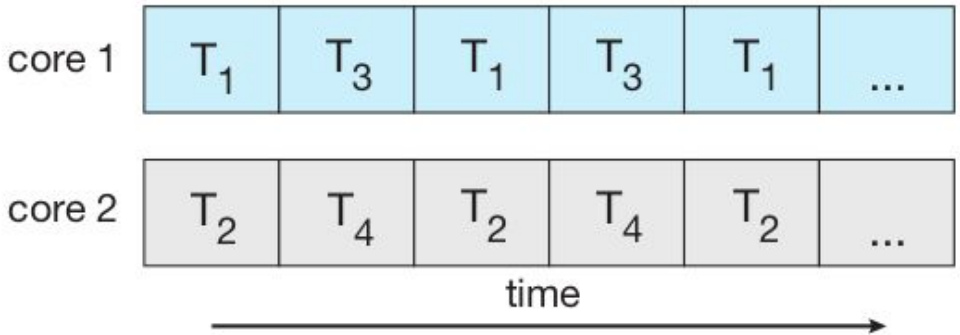
Współbieżność i równoległość



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Współbieżność (ang. *concurrency*) - umożliwia więcej niż jednemu zadaniu być wykonywanym. Do realizacji współbieżności nie jest wymagany system wielordzeniowy.

Współbieżność i równoległość



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

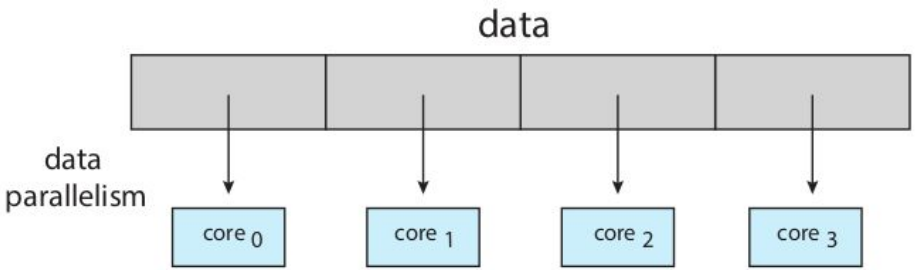
Równoległość (ang. *parallelism*) - umożliwia więcej niż jednemu zadaniu być wykonywanym **JEDNOCZEŚNIE**. Do realizacji równoległości jest wymagany system wielordzeniowy.

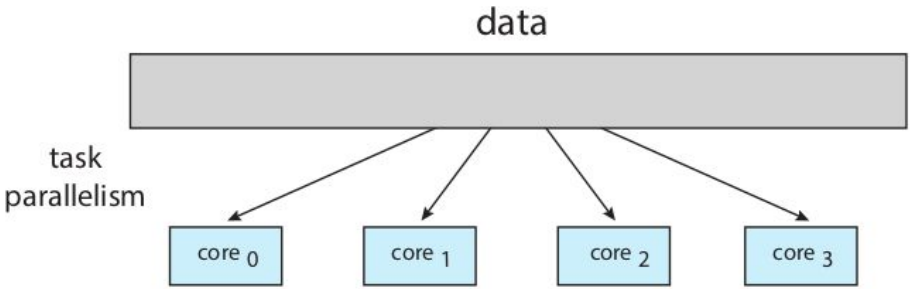
Pytanie: czy może zaistnieć współbieżność bez równoległości ?

Wyzwania programowe

- **Identyfikacja zadań**, w szczególności na zadania niezależne między sobą.
- **Balansowanie** zadaniami celem zrównoważenia obciążenia.
- **Dzielenie danych** między wydzielone zadania.
- **Zależność danych** występująca w szczególności przy następstwie obliczeń.
- **Testowanie i debugowanie** są zdecydowanie trudniejsze niż w programach jednowątkowych.

Typy równoległości





Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Dane dzielone są między procesami/rdzeniami wykonującymi **tego samego typu** operacje.

Przykład: sumowanie zakresów komórek.

Zadania dzielone są między procesami/rdzeniami, a każdy wątek realizuje **unikalną** operację.

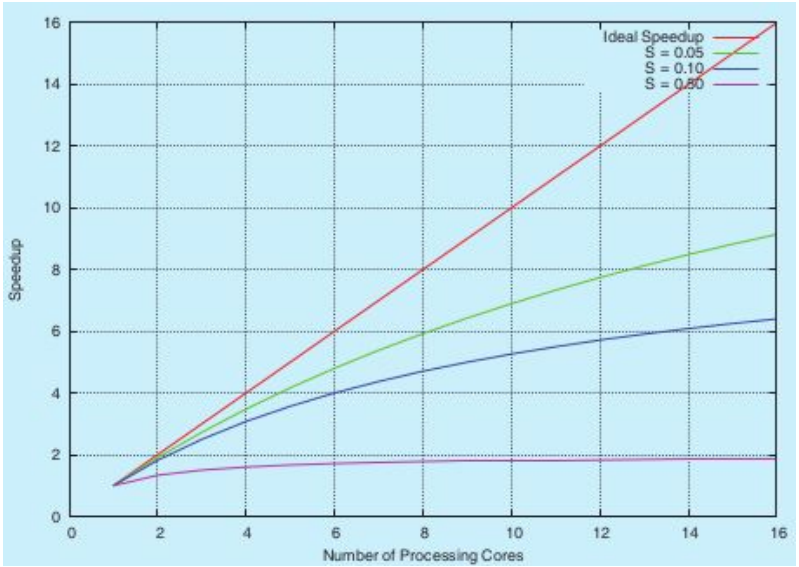
Przykład: jednoczesne wyznaczanie min i max.

Prawo Amdahl'a

Wyraża potencjalny wzrost wydajności obliczeń przez dodanie kolejnych rdzeni obliczeniowych do obsługi aplikacji, która ma dwa komponenty: podlegający i niepodlegający zrównolegleniu.

S - procentowy udział niepodlegającego zrównolegleniu kodu.
N - liczba rdzeni przypisanych do zadania.

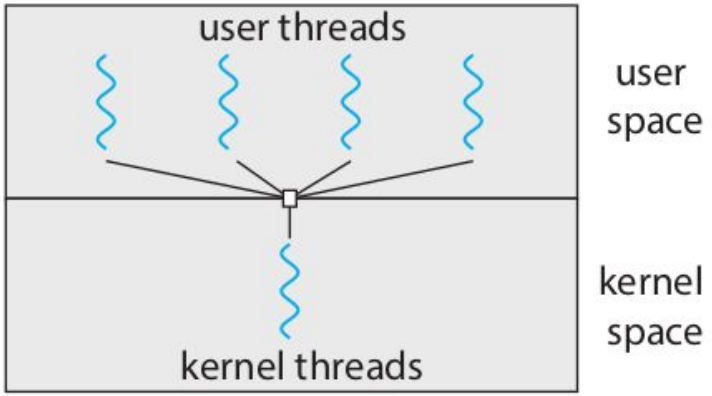
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Model wielowątkowy - Many-to-One

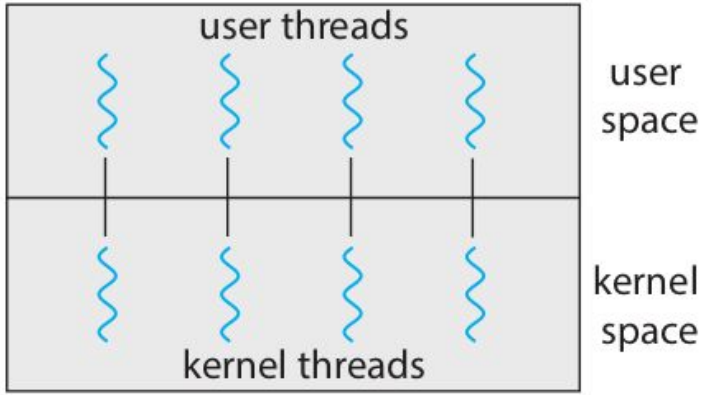
- Zarządzanie wątkami wykonywane jest w przestrzeni użytkownika (przez bibliotekę).
- Zaleta: wydajność.
- Wada 1: zablokowanie całego procesu, jeśli któryś wątek wykona blokujące wywołanie systemowe.
- Wada 2: wątki nie zostaną uruchomione równolegle w systemie wielordzeniowym.
- To tzw. zielone wątki (ang. *green threads*) - można je uruchomić w środowisku nie wspierającym wielowątkowości.
- Przykłady: Solaris, wczesne versje Java.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Model wielowątkowy - One-to-One

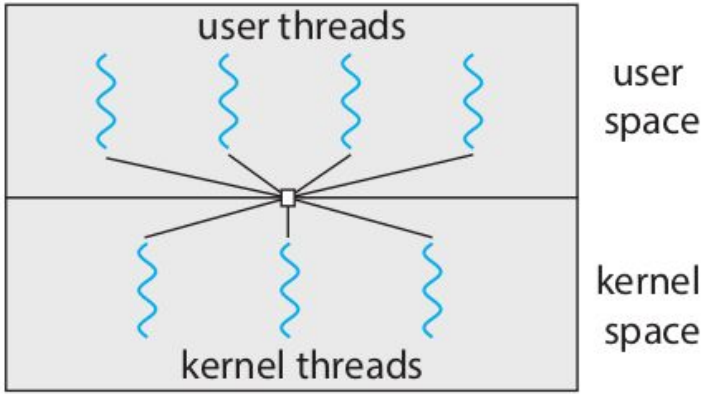
- Model wprowadza niezależną współbieżność, tzn. dany wątek może być realizowany także wtedy, gdy inny wywoła blokującą funkcję systemową.
- Model wprowadza także równoległość.
- Wada: każdy wątek użytkownika tworzy wątek w jądrze, a duża ich liczba może obniżać wydajność systemu.
- Przykłady: Linux, Windows.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Model wielowątkowy - Many-to-Many

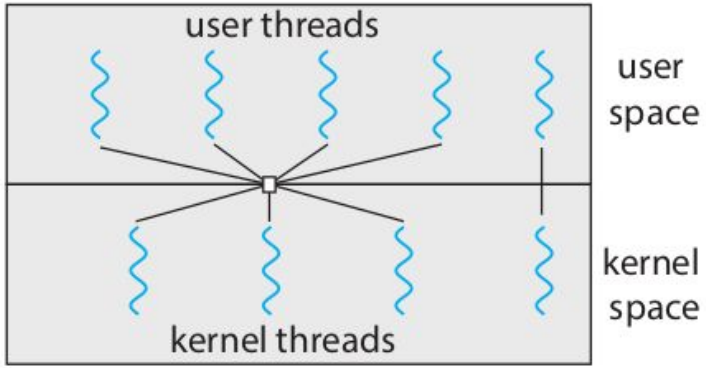
- Model multiplexuje wiele wątków w przestrzeni użytkownika z równą lub mniejszą liczbą wątków w przestrzeni jądra.
- Liczba wątków w przestrzeni jądra może być specyficzna względem aplikacji lub sprzętu.
- Model ten jest pozbawiony wad modeli Many-to-One i One-to-One.
- Trudny w implementacji.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Model wielowątkowy - Two-level

Jak na rysunku obok.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Biblioteki programistyczne dla wątków

- Biblioteki dostarczają API (ang. *Application Programming Interface*) do tworzenia i zarządzania wątkami.
- Podejście 1: Wątki w przestrzeni użytkownika, bez wsparcia ze strony jądra. To oznacza, że wynikiem wywołania funkcji bibliotecznej jest wywołanie funkcji lokalnej, nie systemowej.
- Podejście 2: Wątki w przestrzeni jądra ze wsparciem systemu operacyjnego. To oznacza, że kod i dane biblioteki istnieją w przestrzeni jądra, a wywołanie funkcji jest wywołaniem systemowym.

Biblioteki programistyczne dla wątków (2)

- POSIX Pthreads - przestrzeń użytkownika lub przestrzeń jądra.
- Windows thread library - przestrzeń jądra.
- Java thread API - bezpośrednio w programach Java, tak jak dana implementacja JVM zależna od hostującego systemu operacyjnego.

- POSIX i Windows - dane zadeklarowane globalnie są współdzielone między wątkami procesu.
- Java - nie ma danych globalnych, dostęp do współdzielonych danych musi zostać nadany.

Strategie tworzenia wątków

Strategia asynchroniczna - wątek tworzy wątek potomny i następnie kontynuuje swoje działanie. Oba wątki działają współbieżnie i niezależnie.

Zastosowanie:

- Serwery wielowątkowe.
- Responsywny interfejs użytkownika.

Strategia synchroniczna - wątek tworzy wątki potomne i przechodzi w stan oczekiwania na zakończenie wykonywania ich zadań. O ile wątki potomne działają współbieżnie, to wątek macierzysty po prostu czeka.

Zastosowanie:

- Obliczenia z przesłaniem zadań cząstkowych i oczekiwaniem na wyniki.

Pula wątków

Powody:

- Tworzenie wątków zajmuje pewien czas (mniejszy niż procesów potomnych), a może mogą być wykorzystane ponownie.
- Brak kontroli liczby powstających wątków może doprowadzić do przeciążenia zasobów (procesor, pamięć) systemu.

Rozwiązanie: **pula wątków (ang. *thread pool*)**

- Utworzenie zadanej liczby wątków.
- Umieszczanie wątków w puli wątków.
- Wątki oczekują na przydzielenie zadania.
- Serwer otrzymuje żądanie.
- Serwer przekazuje żądanie do puli wątków.
- Jeśli w puli jest wolny wątek, przejmuje on żądanie i zajmuje się jego obsługą.
- Jeśli brak jest wolnych wątków w puli, zadanie jest kolejkowane.
- Po zakończeniu obsługi danego żądania wątek wraca do puli i oczekuje na nowe.
- Pula wątków najlepiej działa, gdy zadania obsługiwane są asynchronicznie.

Rozmiar puli wątków

Rozmiar puli wątków może być zależny od:

- Liczby rdzeni procesora.
- Ilości fizycznej pamięci RAM.
- Może być też dynamicznie zmieniany w zależności od aktualnie działających wątków (obserwując ich obciążenie).

Wywołania systemowe: `fork()` oraz `exec()`

Problem: czy po wywołaniu przez wątek funkcji systemowej fork() proces potomny duplikuje wszystkie wątki, czy nowy proces jest jedno-wątkowy ?

Odpowiedź: sprawdzić i odpowiedź przedstawić na forum UPEL.

Działanie: wywołanie `exec()` spowoduje zastąpienie całego procesu i wszystkich wątków.

Przypadek I: jeśli `exec()` wywołany jest zaraz po `fork()`, wówczas duplikowanie wszystkich wątków nie jest potrzebne, bo i tak proces zostanie zastąpiony w funkcji `exec()`.

Przypadek II: jeśli `exec()` wywołany jest później, dany `fork()` powinien zduplikować wszystkie wątki.

Obsługa sygnałów

Procedura obsługi sygnałów:

- Sygnał jest generowany przez zdarzenie.
- Sygnał jest dostarczany do procesu.
- Proces musi obsłużyć sygnał.

Sygnał synchroniczny: dzielenie przez 0, nielegalny dostęp do pamięci.

Sygnał asynchroniczny: wciśnięcie np. [ctrl+c].

Zagadka: który wątek otrzyma sygnał ?

Obsługa sygnału:

- **domyślna obsługa sygnału** - jeśli brak zdefiniowanej obsługi sygnału, zajmuje się nią jądro systemu operacyjnego (sygnał może być zignorowany lub zakończyć działanie programu),
- **zdefiniowana przez użytkownika obsługa sygnału**.

Sygnał a program wielowątkowy

Gdzie dostarczyć sygnał ?

- Do wątku, do którego sygnał pasuje.
- Do każdego wątku w procesie.
- Do wybranych wątków w procesie.
- Wybrać jeden wątek do przechwytywania wszystkich sygnałów danego procesu.

Wysyłanie sygnału do procesu:

```
kill(pid_t pid, int signal)
```

Przechwyci go pierwszy nieblokujący wątek.

Wysyłanie sygnału do wybranego wątku:

```
pthread_kill(pthread_t tid, int signal)
```

Dubrownik / Chorwacja