Proces vs. program

Proces to program, który jest wykonywany

- jest dynamiczny, jest uruchomioną instancją programu
- jest aktywny, wykonuje się w pamięci, zużywając zasoby systemowe
- posiada bieżący stan wykonywania, w tym licznik programu, stos, rejestry i inne zasoby systemowe; licznik programu wskazuje następną instrukcję do wykonania

Program to zestaw instrukcji, które mają na celu wykonanie określonego zadania

- jest statyczny, to plik (zbiór bitów) przechowywany na dysku
- jest bierny, nie wykonuje się
- nie jest procesem

Procesy: zadania systemu operacyjnego

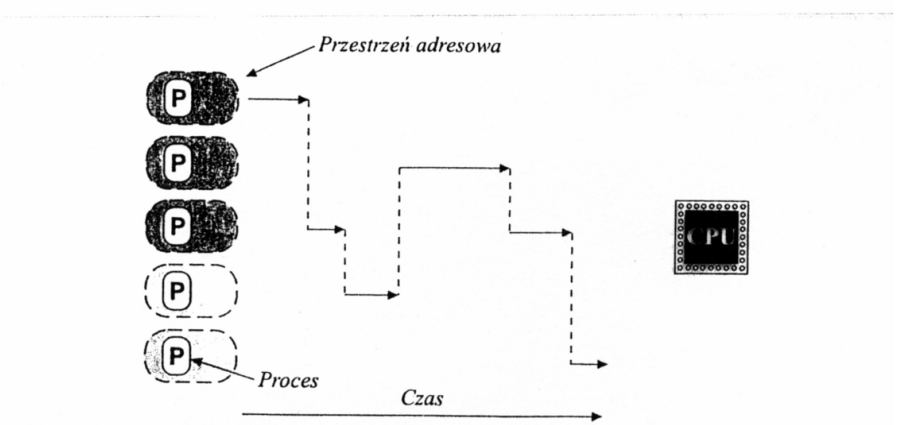
System operacyjny odpowiada za wykonywanie następujących czynności:

- tworzenie i usuwanie procesów
- zarządzanie stanem procesów (wstrzymywanie i wznawianie procesów)
- dostarczanie mechanizmów komunikacji między procesami
- synchronizacja procesów, zapobieganie problemom takim jak zakleszczenia i wyścigi
- obsługa przerwań i sygnałów
- zapewnienie ochrony i bezpieczeństwa, izolacja procesów i zarządzanie uprawnieniami dostępu do zasobów
- zarządzanie przydziałem pamięci dla procesów, wirtualizacja pamięci i ochrona dostępu
- monitorowanie działania procesów i dostarczanie narzędzi do diagnostyki

Procesy: klasyfikacja

- **Rola:**
  - procesy systemowe wykonują kod systemu
  - procesy użytkowników wykonują programy użytkowników
- **Sposób interakcji:**
  - procesy interaktywne wymagają bieżącej interakcji z użytkownikiem (np. terminal, aplikacja GUI)
  - procesy wsadowe wykonują zadania bez bieżącej interakcji z użytkownikiem
- **Tryb działania:**
  - procesy pierwszoplanowe, związane z bieżącą interakcją
  - procesy działające w tle (np. demony, usługi)

Współbieżność (pseudo-paralelizm) - w jednoprocesorowym systemie z podziałem czasu w każdej chwili wykonuje się tylko jeden proces, ale z uwagi na przełączanie kontekstu powstaje wrażenie równoczesnej pracy wielu procesów.



Rysunek 3-1. Tradycyjny system UNIX — jeden procesor z jednowątkowym procesem

Przetwarzanie równoległe – jednoczesne wykonywanie wielu procesów na różnych procesorach, czyli ich rzeczywiste równoczesne wykonywanie.

Tworzenie procesów

Zdarzenia powodujące powstawanie procesów

- inicjalizacja systemu
- wykonanie przez działający proces funkcji systemowej tworzącej procesy
  - Unix/Linux: `fork()`
  - Windows WinAPI: `CreateProcess()`
- żądanie użytkownika w systemie interaktywnym
- inicjacja zadania wsadowego w systemach wsadowych

Zakończenie procesu

Zdarzenia powodujące zakończenie działania procesów:

Dobrowolne:

- normalne wyjście po zakończeniu pracy
  - Unix/Linux: `exit()`
  - Windows WinAPI: `ExitProcess()`
- wyjście zakończone błędem, gdy proces wykryje sytuację uniemożliwiającą dalsze wykonywanie

Niedobrowolne:

- krytyczny błąd, w wyniku wykonania niepoprawnej instrukcji
- zabicie przez inny (uprawniony) proces, poprzez wywołanie funkcji systemowej
  - Unix/Linux: `kill()`
  - Windows WinAPI: `TerminateProcess()`

Hierarchia procesów

- proces, który tworzy nowy proces, nazywany jest procesem **macierzystym** (*parent*) lub **nadrzędnym**, a nowo utworzony proces to proces **potomny** (*child*)
- w UNIX/Linux procesy tworzą wyraźną hierarchię relacji rodzic-potomek (drzewo procesów), pierwszy proces (korzeń) startuje podczas inicjacji systemu (np. `init`, `systemd`), a wszystkie pozostałe są jego potomkami bezpośrednimi lub pośrednimi
- niezależnie od tej hierarchii proces należy do jednej **grupy procesów**, której członkowie mogą wspólnie odbierać sygnały. Każdy proces może mieć własną procedurę obsługi sygnału albo go zignorować. Domyślnie proces potomny należy do tej samej grupy procesów co jego rodzic
- nie wszystkie systemy stosują hierarchię w takim samym stopniu: w Windows proces nadrzędny uzyskuje **uchwyt** do potomka, za pomocą którego może nim zarządzać; uchwyt można jednak przekazać innemu procesowi.

Hierarchia procesów

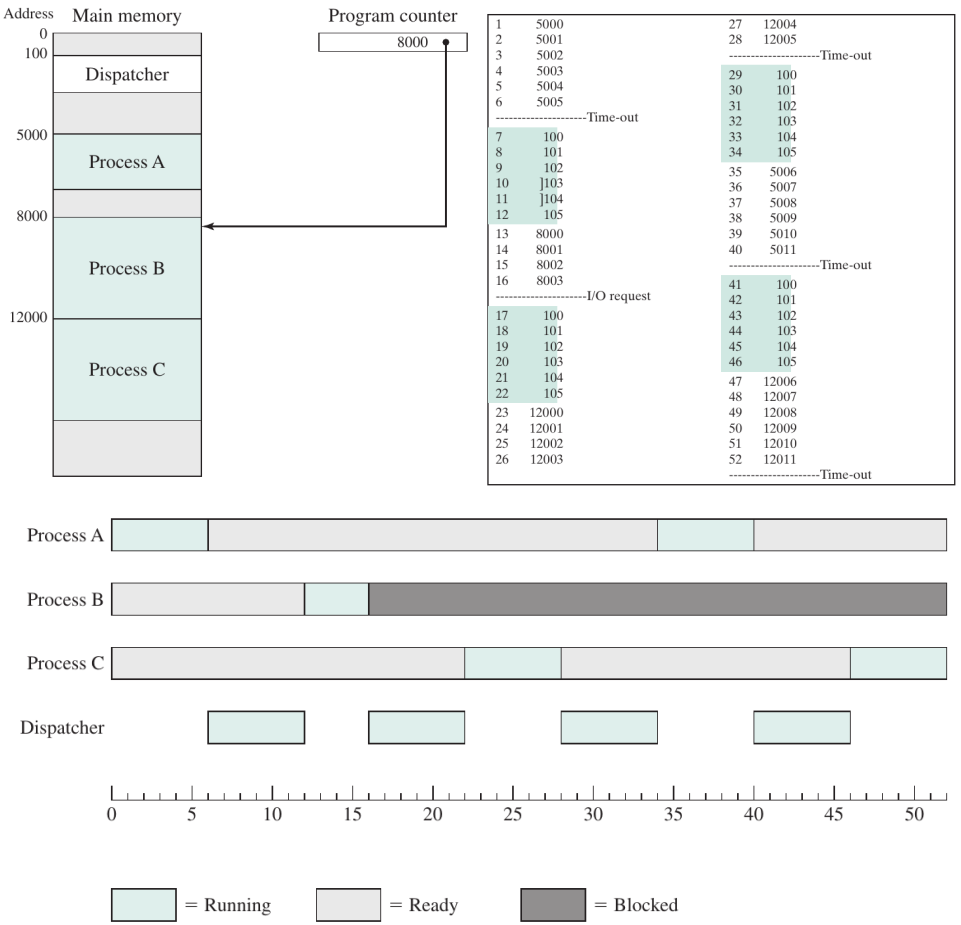
```
$ pstree
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-acpid
        |-atop
        |-atopacctd
        |-avahi-daemon---avahi-daemon
        |-blueman-mechani---2*[{blueman-mechani}]
        |-bluetoothd
        |-cron
        |-gnome-keyring-d-+-ssh-agent
        |-python3
        |-sshd
        |-systemd-+-(sd-pam)
        |         `-gnome-terminal--+-2*[zsh]
        |                           |-zsh---pstree
        |                           `-3*[{gnome-terminal-}]
        `-systemd-journal

$ pstree -s $$
systemd---systemd---gnome-terminal----zsh---pstree
```

Procesy: stany

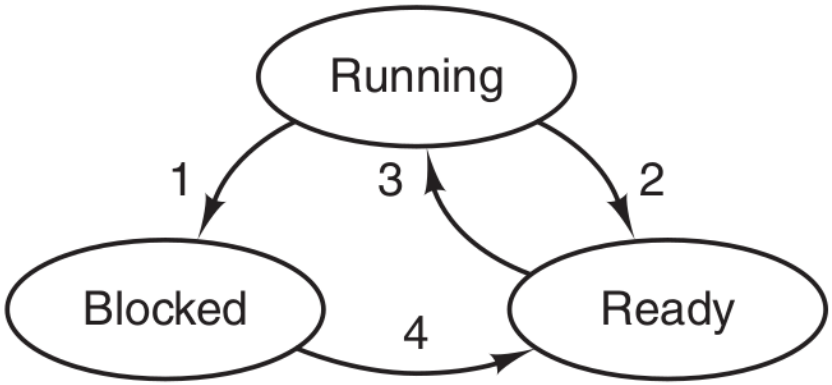
- **gotowy** (*ready*) – proces czeka na przydział procesora
- **bieżący** (*running*) – aktualnie używający CPU, są wykonywane instrukcje
- **zablokowany** (oczekujący, *blocked*) – czeka na wystąpienie jakiegoś zdarzenia (np. zakończenia operacji wejścia-wyjścia)



Przykładowy przebieg procesów w systemie

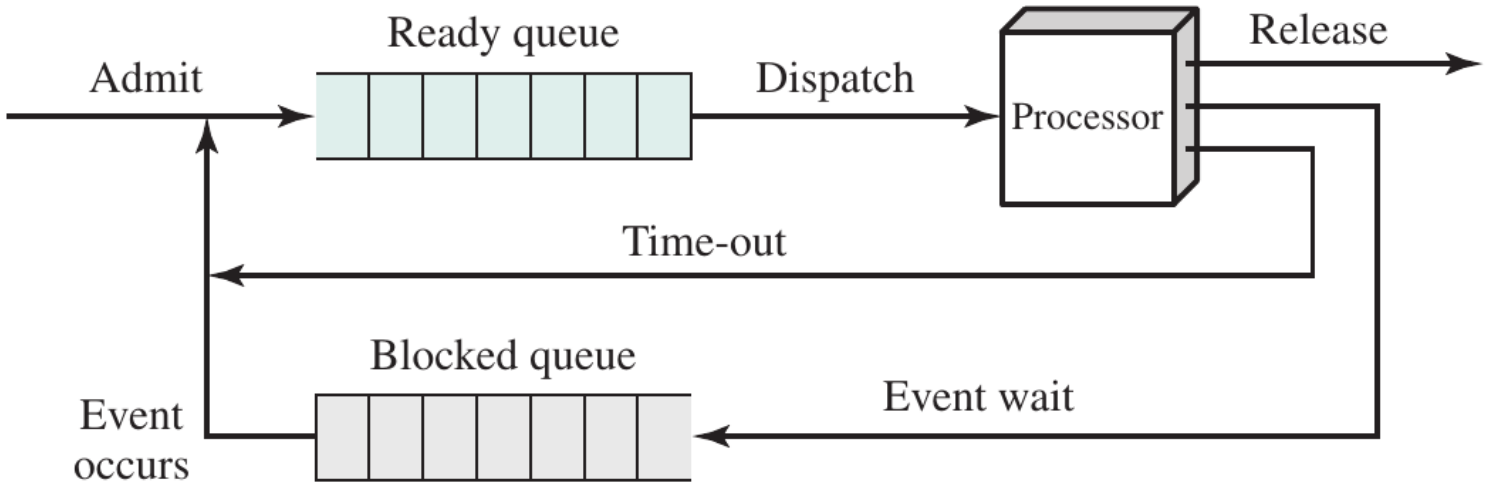


Procesy: zmiana stanu



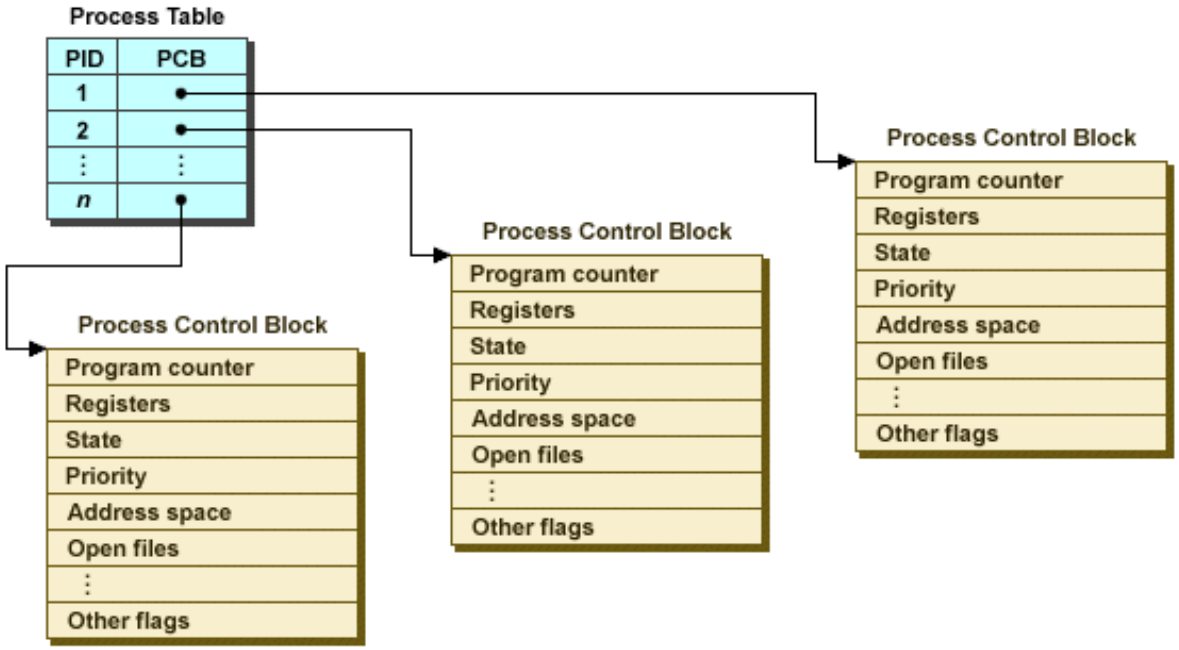
1. proces zablokowany w oczekiwaniu na zdarzenie (np. na zakończenie operacji wejścia)
2. planista wybiera inny proces do wykonania
3. planista wybiera ten proces do wykonania
4. zachodzi zdarzenie odblokowujące proces (wejście staje się dostępne)

Kolejki procesów w stanie gotowości i zablokowania



Procesy: zarządzanie

Tablica procesów - zarządzana przez system operacyjny tablica zawierająca **bloki kontrolne procesów**, po jednej strukturze na proces



Blok kontrolny procesu zawiera wszystkie niezbędne informacje wymagane do zarządzania procesem, wszystko, co musi być zapisane, gdy proces opuszcza stan bieżący, tak aby możliwe było wznowienie jego wykonania.

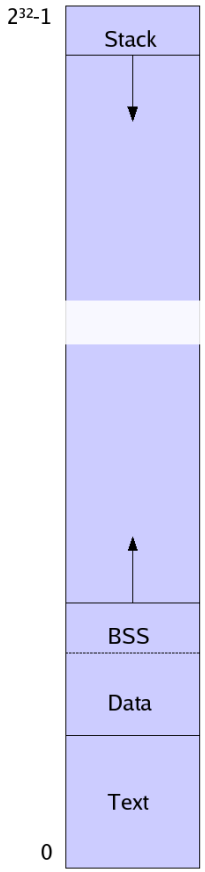
- rejestry procesora, licznik programu, słowo stanu programu, wskaźnik stosu
- informacje o wykorzystanych zasobach (wykaz otwartych plików, ograniczenia pamięci, itp.)
- pozostałe informacje wymagane do zarządzania procesem, planowania przydziału procesora, identyfikacji procesu itp.

Typowe pola bloku kontrolnego procesu

zarządzanie procesem	zarządzanie pamięcią	zarządzanie plikami
rejestry	wskaźnik do informacji segmentu tekstu	katalog główny
licznik programu	wskaźnik do informacji segmentu danych	katalog roboczy
słowo stanu programu	wskaźnik do informacji segmentu stosu	deskryptory plików
wskaźnik stosu		identyfikator użytkownika
stan procesu		identyfikator grupy
priorytet		
parametry szeregowania		
identyfikator procesu		
identyfikator rodzica		
grupa procesów		
sygnały		
czas rozpoczęcia procesu		
wykorzystany czas CPU		
czas CPU procesów potomnych		

Procesy: przestrzeń adresowa

Z procesem związana jest wirtualna przestrzeń adresowa zawierająca segmenty:



- programu (*text*) – kod maszynowy programu, czyli sekwencja instrukcji wykonywanych przez procesor
- danych zainicjowanych (*data*) – zainicjowane zmienne globalne i statyczne
- danych niezainicjowanych (*bss*) – niezainicjowane zmienne globalne i statyczne
- sterty (*heap*) – pamięć alokowana dynamicznie, rośnie wraz z przydzielaniem nowych struktur
- stosu (*stack*) – ramki aktywnych procedur, w których zapisywane są zmienne lokalne oraz adresy powrotu z funkcji

Przykładowe rozmiary segmentów programu:

```
$ size /bin/ps
  text      data     bss     dec     hex filename
114103     16776  135888  266767   4120f  /bin/ps
```

Przykład: polecenie pmap

Komenda pmap (*process memory map*) wyświetla odwzorowanie pamięci procesu (plik /proc/PID/maps).

```
$ pmap -x $$

9719:   zsh
Adres                 KB     RSS  Brudne Tryb  Odwzorowanie
00005fff8366e000      92      92       0 r---- zsh
00005fff83685000     760     760       0 r-x-- zsh
00005fff83743000     108      64       0 r---- zsh
...
00007f398a600000     160     160       0 r---- libc.so.6
00007f398a628000    1620    1440       0 r-x-- libc.so.6
00007f398a7bd000     352     192       0 r---- libc.so.6
...
00007f398aaf6000       8       8       0 r-x--   [ anon ]
00007f398aaf8000       8       8       0 r---- ld-linux-x86-64.so.2
00007f398aafa000     168     168       0 r-x-- ld-linux-x86-64.so.2
00007ffd0a6b5000     356     356     356 rw---   [ stos ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
razem kB           33256   12484    3884
```

Przykład: polecenie ps

Komenda ps (*process status*) pozwala obserwować i określać stan procesów w systemie.

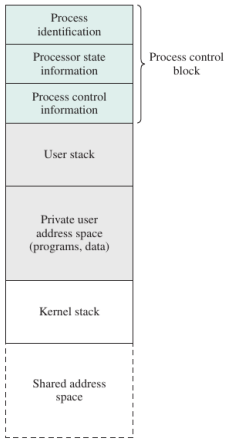
```
$ ps -elf
F S UID          PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1     0  0  80   0 - 55324 ep_pol Nov10 ?        00:00:15 /usr/lib/systemd/systemd
1 S root           2     0  0  80   0 -     0 kthrea Nov10 ?        00:00:00 [kthreadd]
1 S root           4     2  0  60 -20 -     0 worker Nov10 ?        00:00:00 [kworker/0:0H]
1 S root           6     2  0  60 -20 -     0 rescue Nov10 ?        00:00:00 [mm_percpu_wq]
...
4 S avahi        820     1  0  80   0 - 12035 poll_s Nov10 ?        00:00:12 avahi-daemon: running [sc
4 S root         822     1  0  80   0 - 98305 poll_s Nov10 ?        00:00:03 /usr/libexec/accounts-dae
4 S rtkit        823     1  0  81   1 - 45995 poll_s Nov10 ?        00:00:00 /usr/libexec/rtkit-daemon
4 S root         824     1  0  80   0 -  6329 hrtime Nov10 ?        00:00:00 /usr/sbin/smartd -n -q ne
...
0 S jkob        1987  1842  0  80   0 - 30643 poll_s Nov10 pts/2    00:00:00 /bin/bash
...
4 R root       32476 25560  0  80   0 - 36787 -      15:42 pts/22   00:00:00 ps -elf
```

S - stan procesu, UID - identyfikator użytkownika, PID - identyfikator procesu, PPID - identyfikator procesu macierzystego, PRI - priorytet, NI - wartość nice, TIME - całkowity czas CPU użyty przez proces, CMD - polecenie uruchamiające proces

Stos jądra i stos użytkownika w modelu procesu

W typowym systemie komputerowym proces posiada dwa stosy:

- stos użytkownika używany do wykonywania funkcji programu w trybie użytkownika
- stos jądra używany podczas wykonywania kodu jądra



Kod jądra systemu operacyjnego oraz dane znajdują się we współdzielonej przestrzeni adresowej wszystkich procesów. Wywołanie systemowe i obsługa przerwań powodują przełączenie ze stosu użytkownika na stos jądra.

Wątek to jednostka wykonania działająca w ramach procesu.

Wątki tego samego procesu współdzielą jego przestrzeń adresową oraz zasoby, takie jak kod, dane globalne i otwarte pliki.

Wątki posiadają własny:

- licznik programu, wskazujący następną instrukcję do wykonania
- stos z historią wywołań procedur
- rejestry, przechowujące wartości zmiennych lokalnych
- stan (analogiczny do stanu procesu): gotowy, bieżący, zablokowany lub zakończony

Wątki współdzielą w ramach procesu:

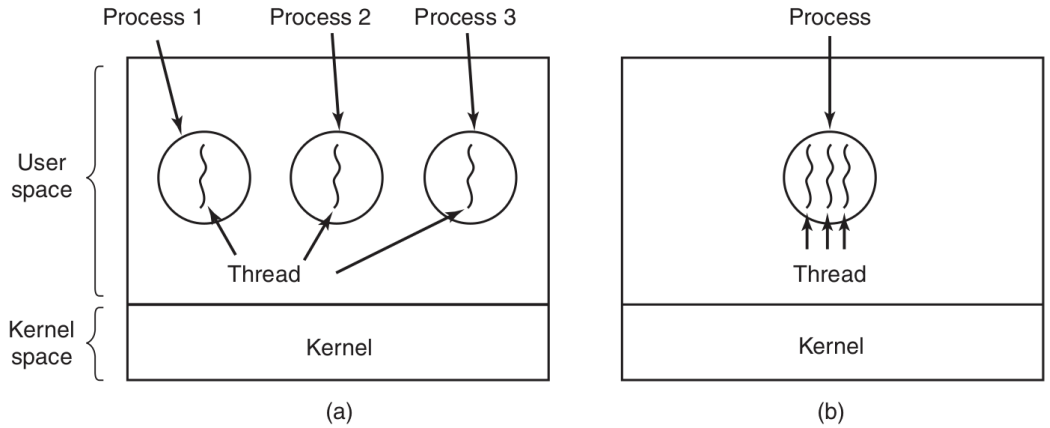
- przestrzeń adresową
- zmienne globalne
- otwarte pliki
- zasoby systemowe przypisane do procesu
- sygnały i procedury obsługi sygnałów przypisane do procesu
- uprawnienia przypisane do procesu

Procesy i wątki

- Procesy grupują zasoby, zaś wątki są jednostkami wykonawczymi pracującymi na CPU.
- Procesy są zwykle od siebie odizolowane, a ich współpraca wymaga mechanizmów komunikacji międzyprocesowej. Zarówno procesy, jak i wątki mogą współpracować lub konkurować o zasoby.
- Wątki czasem nazywane są lekkimi procesami, ponieważ posiadają pewne cechy procesu, ale są lżejsze pod względem zarządzania i przełączania kontekstu.

Wielowątkowość – możliwość występowania wielu wątków w procesie. Pozwala lepiej wykorzystywać zasoby procesora i ułatwia realizację zadań współbieżnych w ramach jednego procesu.

Procesy i wątki

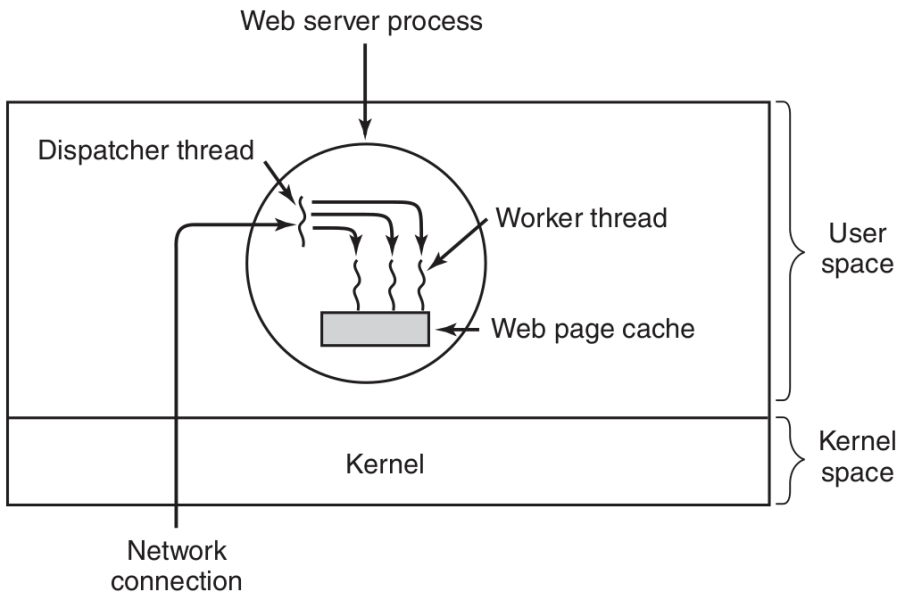


Proces	Wątek
przestrzeń adresowa	licznik programu
zmienne globalne	rejestry
otwarte pliki	stos
procesy potomne	stan
zaległe alarmy	
sygnały i procedury obsługi sygnałów	
informacje dotyczące statystyk	

Zalety wątków

- przełączanie procesora między wątkami w procesie trwa krócej niż przełączanie między procesami
- tworzenie i kończenie wątku w istniejącym procesie zajmuje o wiele mniej czasu niż tworzenie i niszczenie procesu (nawet 10-100 razy szybsze)
- lepsza realizacja przetwarzania współbieżnego na maszynach wieloprocesorowych
- sprawniejsza komunikacja między wątkami niż między procesami (wątki współdzielą pamięć procesu), komunikacja nie wymaga angażowania jądra

Przykład zastosowania wątków: serwer WWW



Dyspozytor

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

Wątek roboczy

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Dyspozytor

- odbiera kolejne żądania i przekazuje je do wątków roboczych

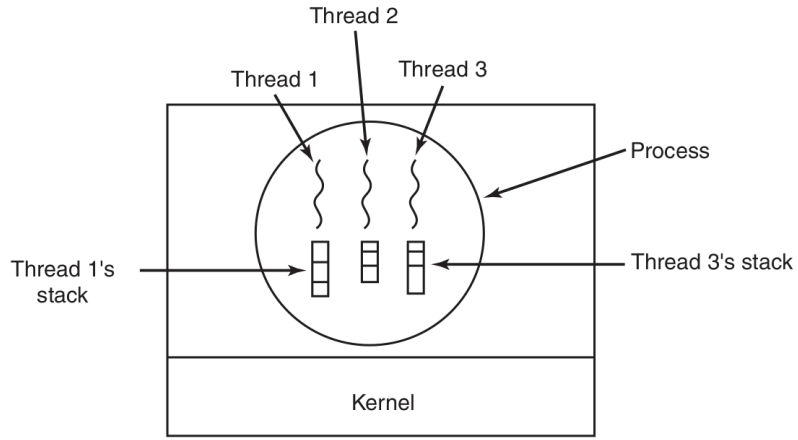
Wątek roboczy

- szuka strony w pamięci podręcznej; w razie braku odczytuje ją z dysku
- odsyła stronę klientowi

Zalety

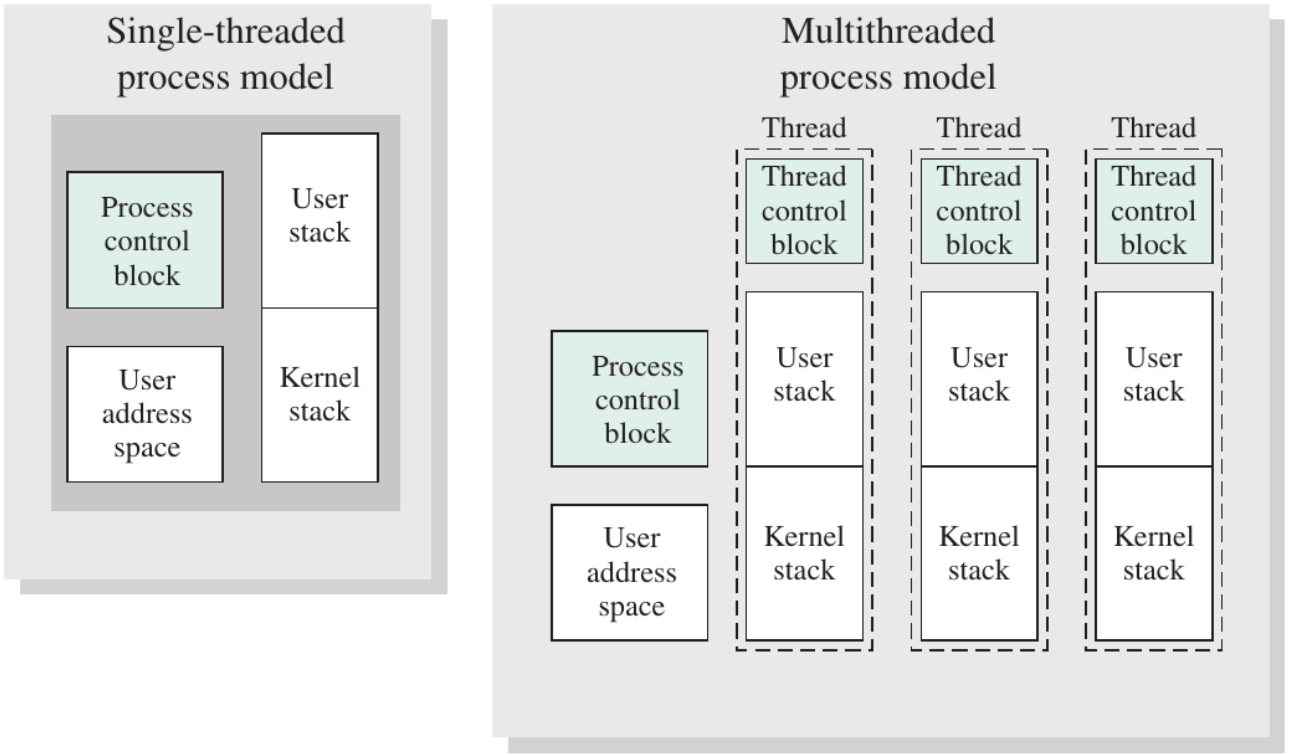
- współbieżna obsługa wielu żądań
- oczekiwanie na I/O jednego wątku nie zatrzymuje innych, jeśli są dostępne
- wątki współdzielą pamięć podręczną i są tańsze w obsłudze niż procesy

Stos procesu



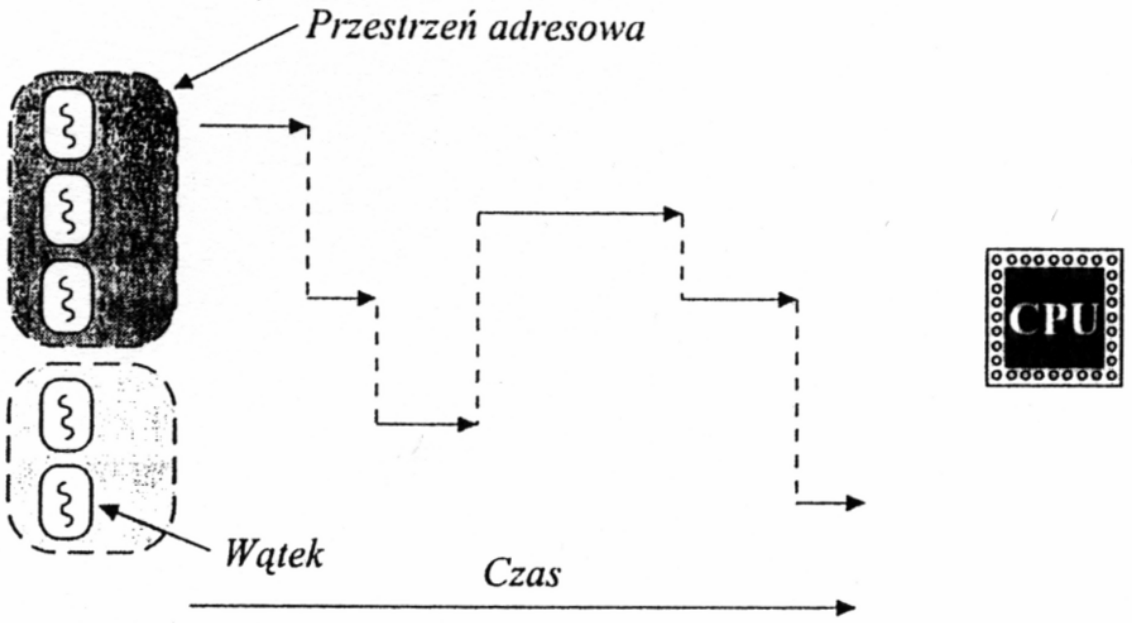
- każdy wątek posiada własny stos
- stos zawiera po jednej ramce dla każdej procedury, która została uruchomiona, a z której jeszcze nie nastąpił powrót
- ramka zawiera zmienne lokalne procedury i adres powrotu
- każdy wątek może uruchamiać różne procedury i posiadać odmienną historię wywołań

Model procesu jednowątkowego i wielowątkowego



Każdy wątek obsługiwany przez jądro ma własny stos jądra. W procesie wielowątkowym każdy wątek ma także własny stos użytkownika.

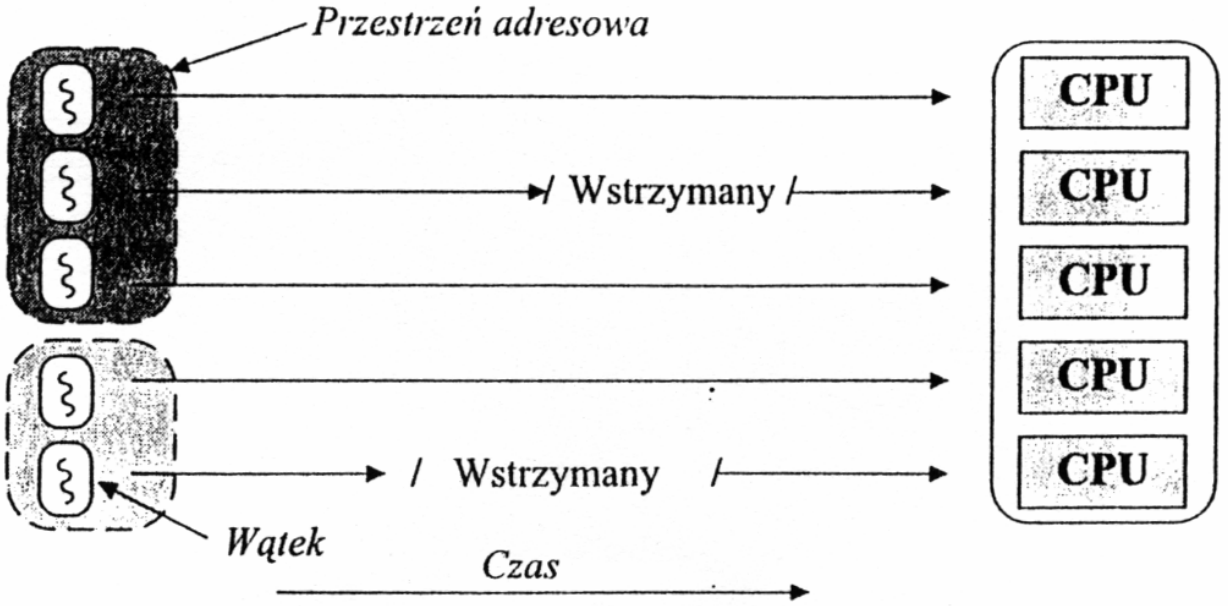
Procesy wielowątkowe w systemie z jednym CPU



Rysunek 3-2. Procesy wielowątkowe w systemie z jednym procesorem

Niektóre procesory obsługują sprzętową wielowątkowość (SMT), np. Intel Hyper-Threading, która pozwala jednemu rdzeniowi wykonywać współbieżnie instrukcje z dwóch wątków.

Procesy wielowątkowe w wieloprocesorze

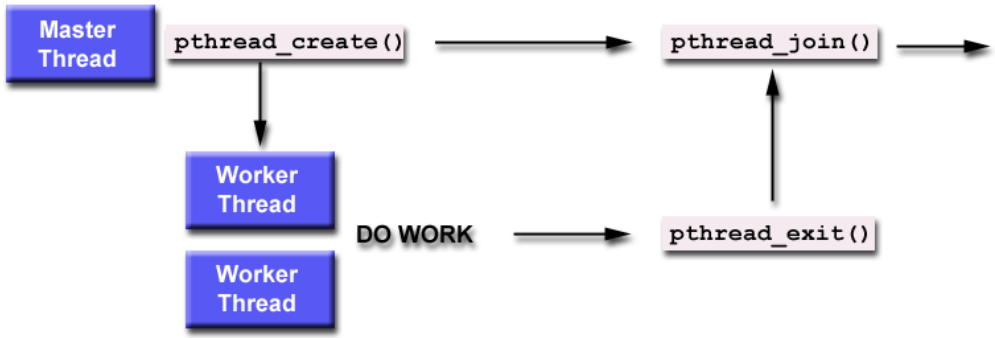


Rysunek 3-3. Procesy wielowątkowe w wieloprocesorze

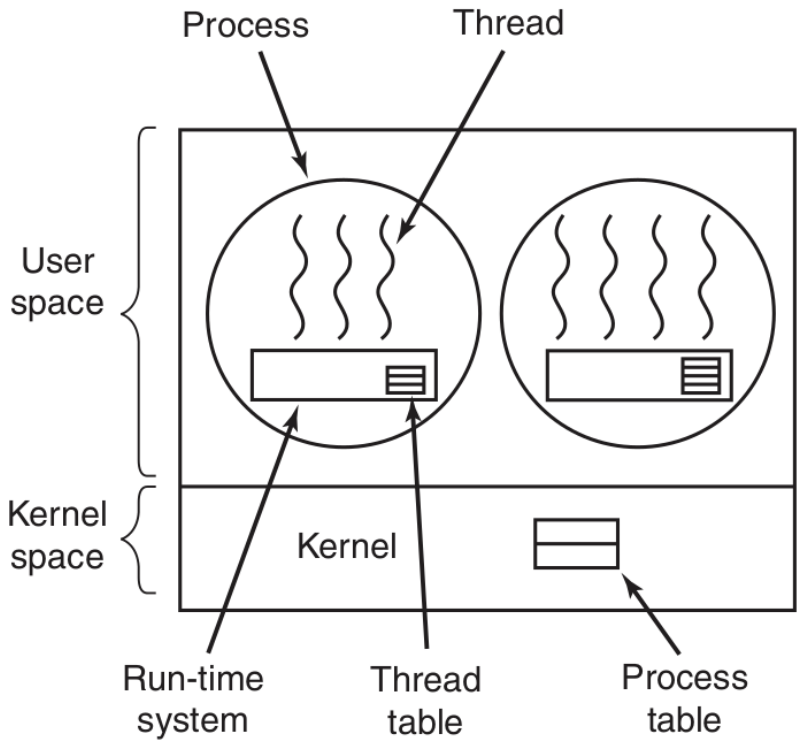
Nowoczesne procesory mają wiele rdzeni, co pozwala na jednoczesne wykonywanie wielu wątków w sposób rzeczywiście równoległy

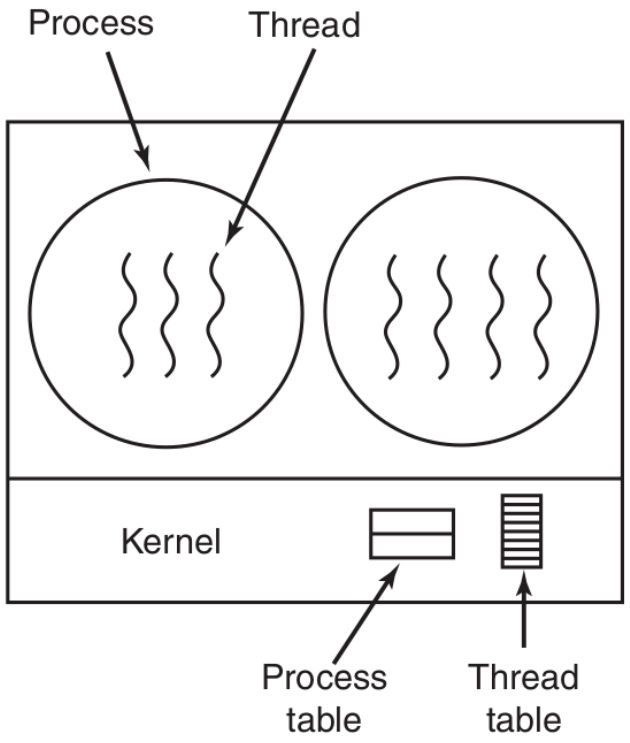
Zarządzanie wątkami

- nowo utworzony proces zazwyczaj startuje z pojedynczym wątkiem
- wątek może utworzyć nowy wątek (funkcja `thread_create`), przekazując mu procedurę do wykonania
- wątek po wykonaniu pracy zgłasza zamknięcie (funkcja `thread_exit`)
- wątek może oczekiwać na zakończenie innego wątku (funkcja `thread_join`), jest wówczas w stanie zablokowanym
- wątek może dobrowolnie zrzec się CPU (funkcja `thread_yield`), tak aby dać innym wątkom szansę na wykonanie



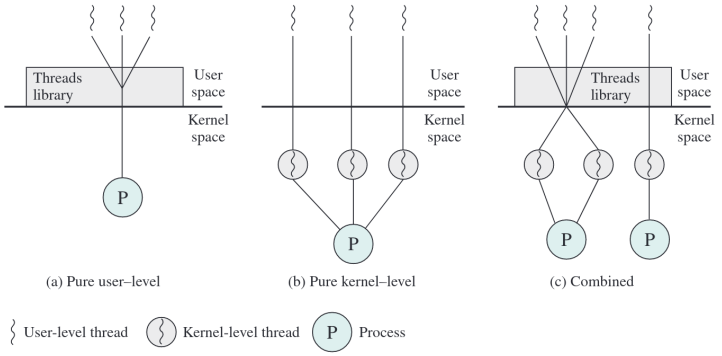
Wątki poziomu użytkownika i poziomu jądra





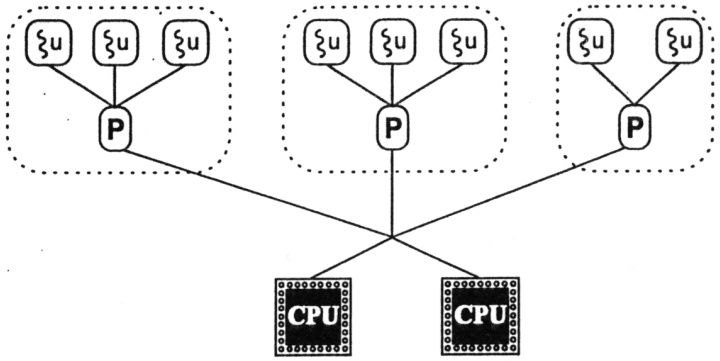
Rodzaje wątków

- **wątki poziomu użytkownika** (*user-level threads*) - zarządzane przez bibliotekę w przestrzeni użytkownika, np. GNU Portable Threads
- **wątki poziomu jądra** (*kernel-level threads*) zarządzane bezpośrednio przez jądro systemu operacyjnego, np. PThreads (p-wątki, wątki zgodne z POSIX), Windows Thread API, Native POSIX Thread Library (NPTL)
- **wątki hybrydowe (mieszane)** (*hybrid threads*) – model, w którym biblioteka zarządza wieloma wątkami użytkownika, mapując je na pulę wątków poziomu jądra (model M:N)

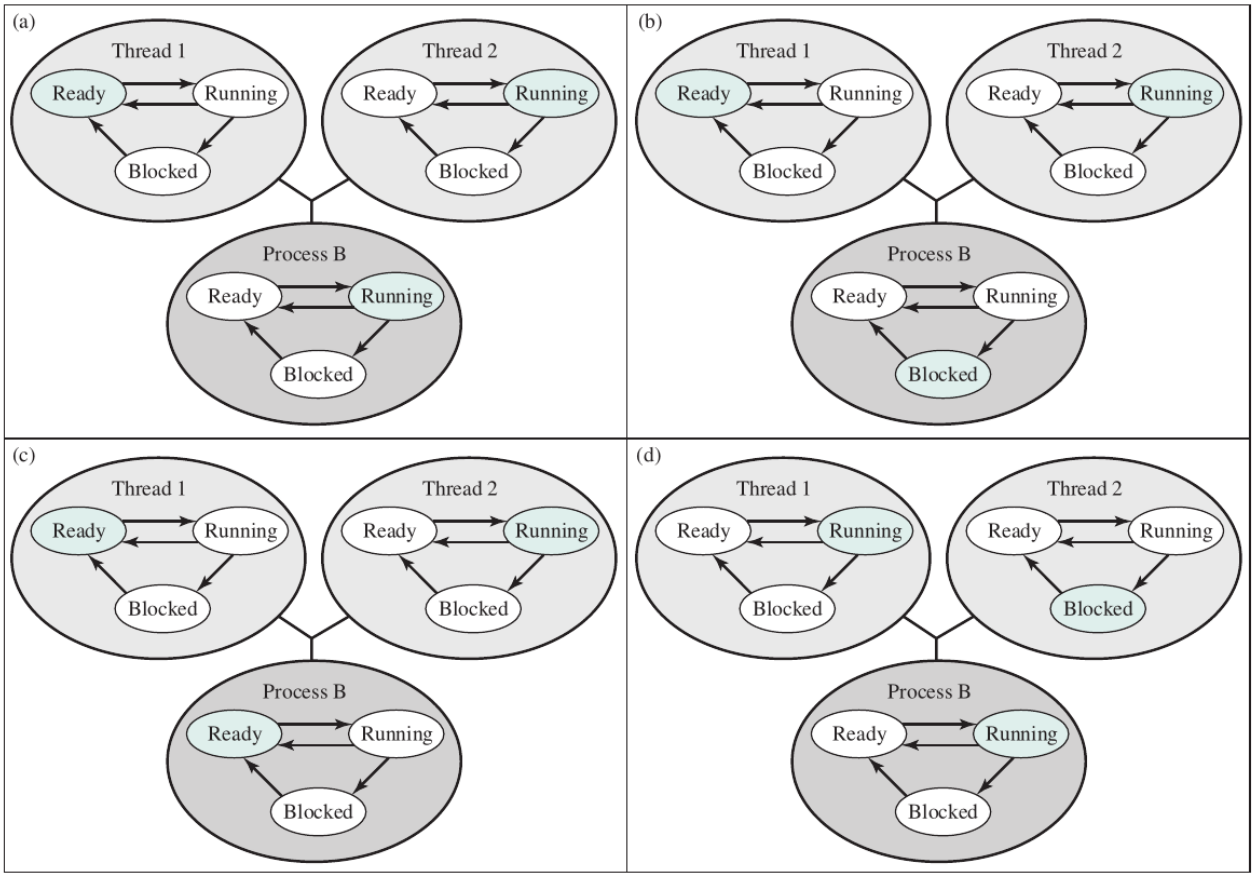


Wątki poziomu użytkownika

- wątki poziomu użytkownika (*user-level threads*, ULT) - wątki działające w przestrzeni użytkownika są implementowane i zarządzane przez bibliotekę
- każdy proces przechowuje własną **tablicę wątków**, wskazującą na zasoby wątków związanych z procesem (licznik programu, rejestry, stos)
- kontekst wątku jest zapamiętywany i odtwarzany z poziomu użytkownika, bez angażowania jądra
- jądro zarządza procesem (nie wie o istnieniu wątków użytkownika); wywłaszczając proces użytkownika, wywłaszcza związane z nim wątki



Stany wątków użytkownika a stany procesu



operacja blokująca wątku użytkownika blokuje proces (biblioteka nie działa i nie aktualizuje stanu wątków)

proces wyczerpał kwant czasu lub został wywłaszczony

wątek 2 blokuje się na rzecz wątku 1

Planowanie wątku odbywa się na poziomie użytkownika (jądro nie jest świadome stanu wątków), a planowanie procesu realizowane jest przez jądro systemu operacyjnego.

Zalety wątków użytkownika

- **Błyskawiczne przełączanie kontekstu:** przełączanie wątków nie wymaga przejścia do trybu jądra i odbywa się w przestrzeni użytkownika
- **Wydajność:** wątki użytkownika są wydajne, nie zużywają zasobów jądra, tworzenie wątku sprowadza się do alokacji małej struktury danych w przestrzeni użytkownika
- **Własny algorytm zarządzania:** każdy proces może mieć indywidualny schemat zarządzania wątkami, dostosowany do potrzeb aplikacji
- **Niezależność od systemu:** mogą być implementowane w systemach operacyjnych, które nie wspierają wielowątkowości na poziomie jądra

Problemy wątków użytkownika

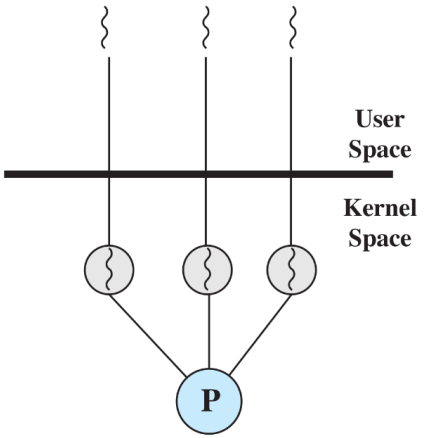
- **Ograniczenie zrównoleglenia:** biblioteka szereguje wątki, jądro procesy; tylko jeden wątek procesu może być wykonywany w danym momencie nawet na maszynie wieloprocesorowej
- **Blokowanie procesu:** operacje blokujące (wywołania systemowe, błędy strony) powodują zablokowanie całego procesu, więc i wszystkich wątków poziomu użytkownika w procesie
- **Kosztowne wywołania systemowe:** wywołanie systemowe powoduje synchroniczne przejście do trybu jądra; przejście i obsługa w jądrze wiążą się z narzutem
- **Monopolizacja CPU:** uruchomiony wątek zajmuje CPU, nie dając możliwości działania innym wątkom, dopóki nie zrzeknie się procesora (*yield*)

Osłanianie (ang. *jacketing*) - technika polegająca na owinięciu wywołań systemowych wątku użytkownika w funkcje, które zarządzają blokowaniem i odblokowywaniem innych wątków w procesie.

Wątki poziomu jądra

- wątki poziomu jądra (KLT, *kernel-level threads*) są zarządzane bezpośrednio przez jądro systemu operacyjnego
- jądro utrzymuje tablicę wątków, podobnie jak tablicę procesów
- jądro jest świadome istnienia wątków i może nimi zarządzać, planuje wątki na procesorach, przełącza je, tworzy i niszczy
- aplikacje i programiści korzystają z interfejsów API udostępnianych przez system operacyjny, które abstrahują zarządzanie wątkami

Windows API:	`CreateThread()`
POSIX:	`pthread_create()`
C++ 11:	`std::thread`



Zalety wątków poziomu jądra

- **Prawidłowe blokowanie:** w momencie zablokowania wątku system może wznowić wykonywanie innego, gotowego wątku (także z innego procesu)
- **Wykorzystanie wieloprocesowości:** system operacyjny może jednocześnie planować wiele wątków z jednego procesu na wielu procesorach
- **Precyzyjna kontrola:** jądro może precyzyjnie kontrolować dostęp do zasobów i zarządzać priorytetami wątków
- wątki poziomu jądra nie wymagają implementacji nieblokujących wywołań systemowych; w momencie wywołania blokującego inny wątek może być wykonywany

Wady wątków poziomu jądra

- **Kosztowne przełączanie kontekstu:** przełączanie wątków przez jądro jest wolniejsze niż w przypadku wątków użytkownika, ponieważ wymaga przejścia z trybu użytkownika do trybu jądra i z powrotem
- **Większe zużycie zasobów:** tworzenie oraz niszczenie wątków odbywa się przez odwołania do jądra, jest to kosztowniejsze niż w przypadku wątków użytkownika

Procesy i wątki: porównanie

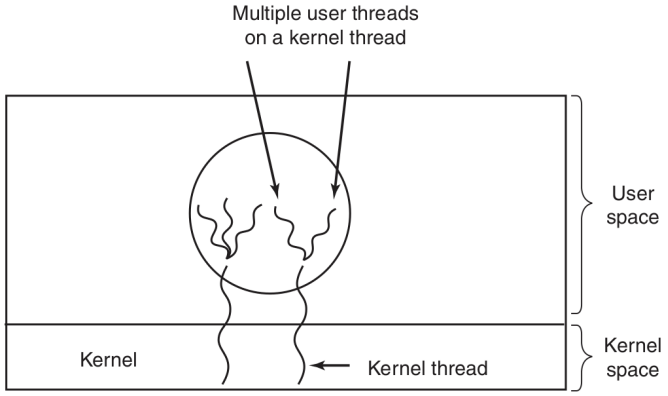
- **Null fork:** czas mierzony od utworzenia procesu lub wątku, przez jego szeregowanie i wykonanie, aż do jego zakończenia (wątku), który wykonuje pustą procedurę (miara obciążenia spowodowanego przez tworzenie procesu (wątku))
- **Signal-Wait:** czas potrzebny, aby proces (wątek) przesłał sygnał oczekiwania innemu procesowi (wątkowi) i otrzymał odpowiedź (miara obciążenia spowodowanego wzajemną synchronizacją)

Czas oczekiwania (μs) na wykonanie operacji przez wątki poziomu użytkownika (ULT), wątki poziomu jądra (KLT) i procesy (P).[1]

	ULT	KLT	P
Null fork	34	948	11300
Signal-wait	37	441	1840

[1] VAX, system uniksopodobny

Hybrydowe implementacje wątków



- podejście mieszane, wykorzystuje wątki poziomu jądra i użytkownika (model M:N)
- aplikacja tworzy wiele wątków poziomu użytkownika (M) i są one mapowane na mniejszą lub równą liczbę wątków poziomu jądra (N)
- jądro systemu zarządza wyłącznie wątkami jądra
- zarządzanie wątkami użytkownika odbywa się w ramach procesu, analogicznie do tradycyjnego modelu wątków poziomu użytkownika

Zalety hybrydowych wątków

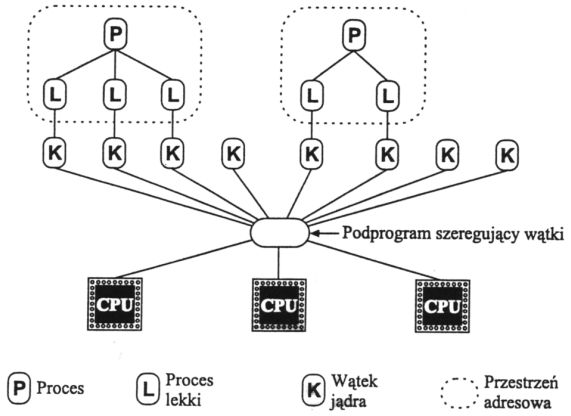
Wątki hybrydowe łączą zalety wątków poziomu użytkownika i wątków poziomu jądra.

- większość prac związanych z zarządzaniem wątkami (tworzenie, synchronizacja) odbywa się na poziomie użytkownika
- programista może dobrać liczbę wątków jądra do konkretnej aplikacji i procesora
- wiele wątków tej samej aplikacji może być wykonywanych równolegle na wielu procesorach
- wątki mogą być blokowane niezależnie, bez blokowania całego procesu

Wadą hybrydowych wątków jest większa złożoność implementacji i zarządzania wątkami, zarówno na poziomie użytkownika, jak i jądra.

Procesy lekkie, LWP

- Proces lekki (*light-weight process*, LWP) jest wątkiem w przestrzeni użytkownika wspieranym przez jądro
- z procesem może być związanych wiele procesów lekkich, z których każdy jest wspierany przez osobny wątek jądra (relacja 1:1)
- procesy lekkie są niezależnie szeregowane, współdzielą przestrzeń adresową, mogą wywoływać blokujące funkcje systemowe bez blokowania procesu, mogą się wykonywać na różnych procesorach



https://www.thegeekstuff.com/2013/11/linux-process-and-threads/

Pojęcie procesów lekkich bywa używane w kilku znaczeniach:

- oryginalnie w Solarisie (Unix) oznacza mapowanie wątku użytkownika na wątek jądra
- w Linuksie proces lekki to jednostka wykonawcza (zadanie) planowana przez jądro. Może być procesem lub wątkiem (oba są reprezentowane przez strukturę `task_struct`), dlatego termin ten bywa używany jako synonim wątku poziomu jądra

Lekkie wątki użytkownika: w systemie Windows `fibers` (włókna) są wątkami wykonywanymi całkowicie w trybie użytkownika. Nie są zarządzane bezpośrednio przez jądro systemu, lecz działają w kontekście wątku jądra.

Tworzenie procesu w UNIX/Linux

Funkcja systemowa `fork()` tworzy nowy proces

- przydziela nowemu procesowi pozycję w tablicy procesów
- przydziela procesowi potomnemu unikatowy identyfikator PID
- tworzy logiczną kopię procesu macierzystego (ew. zapewniając współdzielenie segmentów instrukcji, itp.);
- zwiększa plikom związanym z tym procesem liczniki w tablicy plików i i-węzłów
- przekazuje identyfikator potomka procesowi macierzystemu i wartość zero procesowi potomnemu

Mechanizm **kopiowania przy zapisie** (*copy-on-write*, COW) odkłada w czasie tworzenie rzeczywistej kopii procesu, aż do momentu gdy nastąpi zmiana w pamięci procesu macierzystego lub potomnego. Jest realizowany niezależnie przez podsystem zarządzania pamięcią i system stronicowania (MMU, *Memory Management Unit*)

Tworzenie procesu w UNIX/Linux

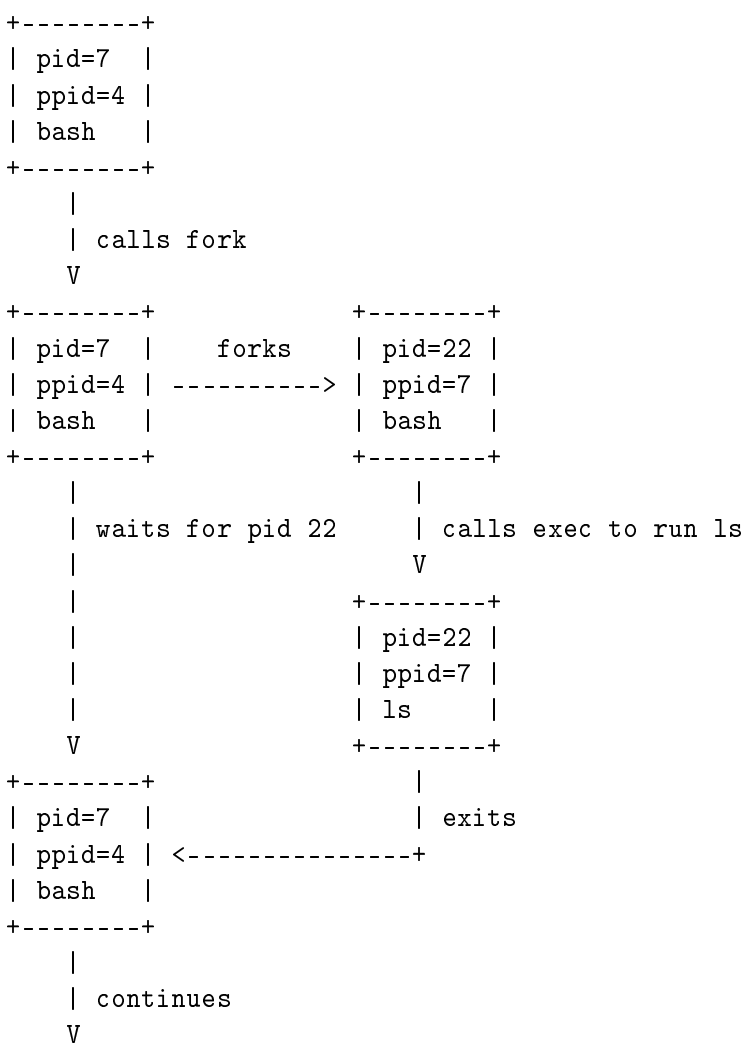
Wywołanie systemowe `exec()` umieszcza w obszarze pamięci bieżącego procesu kopię pliku wykonywalnego

- aktualny proces zaczyna wykonywać nowy program
- zawartość kontekstu poziomu użytkownika staje się niedostępna, z wyjątkiem parametrów `exec()`, które jądro kopiuje ze starej do nowej przestrzeni adresowej
- w bibliotece systemowej wywołanie systemowe `exec()` realizowane jest przez funkcję `execve()` lub inne odmiany tej funkcji

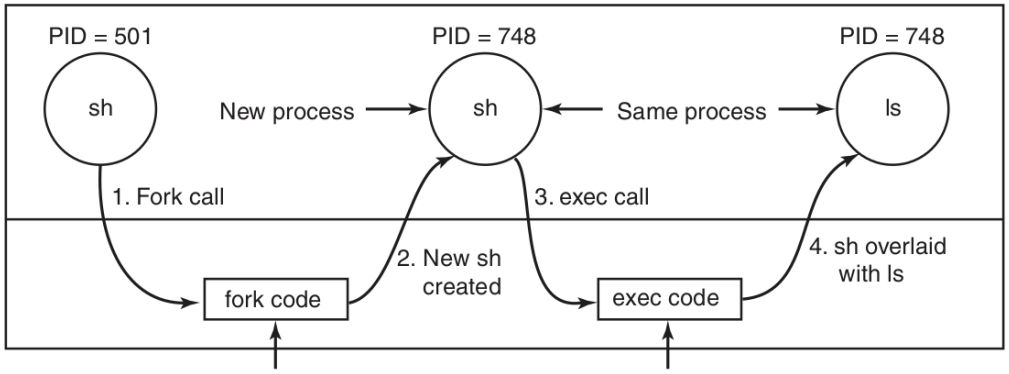
Procesy: tworzenie

Implementacja prostej powłoki

```
while(TRUE) {
   type_command();
   read_command(command, params);

   pid = fork();
   if (pid < 0) {
      printf("Unable to fork()");
      continue;
   }

   if (pid != 0) {
      /* rodzic czeka na potomka */
      waitpid(-1, &status, 0);
   } else {
      /* potomek wykonuje pracę */
      execve(command, params, 0);
   }
}
```



Uruchomienie procesu w powłoce



`fork()` – tworzenie potomka

1. przydziela pamięć dla struktury procesu potomnego
2. wypełnia strukturę procesu potomnego danymi rodzica
3. przydziela pamięć dla stosu i przestrzeni użytkownika procesu potomnego
4. wypełnia przestrzeń użytkownika procesu potomnego danymi procesu rodzica
5. przydziela procesowi potomnemu PID
6. ustawia współdzielenie tekstu z rodzicem
7. kopiuje tablice stron danych i stosu
8. ustawia współdzielenie plików z rodzicem
9. kopiuje rejestry rodzica do dziecka

Obowiązuje kopiowanie przy zapisie (*Copy on Write*, CoW)

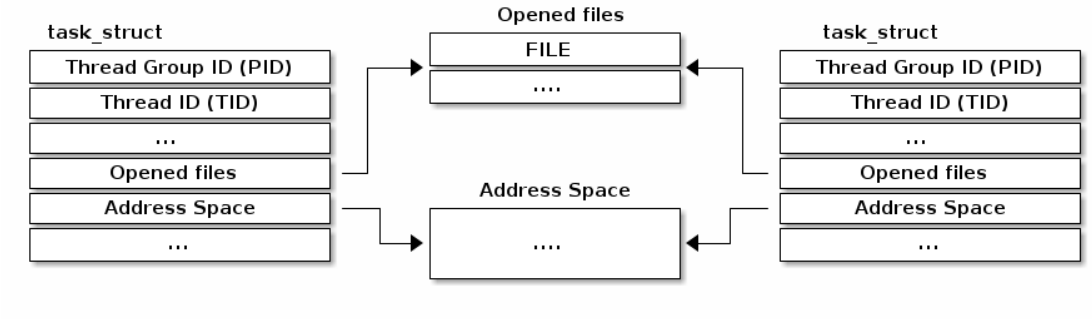
`exec()` – zastąpienie obrazu procesu nowym programem

1. odnajduje program wykonywalny
2. sprawdza uprawnienia wykonania
3. odczytuje i weryfikuje nagłówek programu
4. kopiuje argumenty i łańcuchy środowiska do jądra
5. zwalnia starą przestrzeń adresową
6. przydziela nową przestrzeń adresową
7. kopiuje argumenty i łańcuchy środowiska na stos
8. zeruje sygnały
9. inicjuje rejestry

Procesy: zarządzanie w systemie GNU/Linux

W jądrze Linuksa procesy i wątki są reprezentowane jako **zadania** (*task*) za pomocą struktury danych `task_struct`.

- struktura zadania reprezentuje jednostkę wykonawczą (proces lub wątek)
- zasoby zadania nie są osadzone w strukturze, struktura posiada wskaźniki do zasobów współdzielonych z innymi zadaniami
- jednowątkowy proces użytkownika to pojedyncze zadanie, wielowątkowy proces to wiele zadań (po jednym na wątek), ale dzielących wspólne zasoby



Najważniejsze atrybuty struktury zadań task_struct

- informacje o zadaniu (*task info*): jednoznaczny identyfikator zadania (TID), informacje o związkach rodzic/dziecko pomiędzy zadaniami, identyfikator grupy zadań (TGID), gdzie zadania z pojedynczej grupy TGID reprezentują tradycyjny proces wielowątkowy
- przestrzeń adresowa (*address space*): definiuje przestrzeń wirtualną zadania
- informacje o przydziale procesora (*scheduling info*): status zadania, polityka przydziału procesora, parametry tej polityki (priorytet dynamiczny)
- informacja o programie wykonywalnym (*executable info*): identyfikuje plik, który jest obrazem wykonywanego zadania
- informacja o sygnałach (*signal info*): dane związane z sygnałami i ich obsługą (*signal handler table, pending signal mask, signal queue*)
- dane uwierzytelniające (*credentials*): dane określające prawa i przywileje zadania (UID, GID, maska określająca prawa dostępu do urządzeń)

- informacje księgowe (*accounting info*): czas utworzenia zadania, czas zużyty w trybie jądra i użytkownika, liczba błędów stron, itp.
- ograniczenia zasobów (*resource limits*): parametry określające maksymalną wielkość pliku *core*, limity czasu wykonania zadania, wykorzystania pamięci, liczby otwartych plików, itp.
- informacja o systemie plików (*filesystem info*): domyślna maska określająca prawa dostępu przy tworzeniu plików (*umask*), bieżący katalog
- tablica otwartych plików (*open file table*)
- różne (*miscellaneous*): dodatkowe atrybuty zadania potrzebne do jego prawidłowego wykonywania się, np. terminal sterujący

Jądro systemu GNU/Linux: tworzenie zadania

Wywołanie systemowe `clone()` tworzy nowe zadania (wątki lub procesy)

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Uruchamia funkcję w kontekście nowego wątku, który współdzieli (lub nie) zasoby zadania zależnie od wartości `sharing_flags`

flagi	współdzielone zasoby	gdy ustawiona	gdy nie ustawiona
CLONE_VM	przestrzeń adresowa	współdzielona z rodzicem	osobna kopia przestrzeni adresowej
CLONE_FS	informacje o systemie plików katalog roboczy, główny, umask	wspólny katalog roboczy	niezależny katalog roboczy
CLONE_FILES	tablica otwartych plików	współdzielenie deskryptorów	kopie deskryptorów
CLONE_SIGHAND	tablica obsługi sygnałów	współdzieli	kopiuje tablicę
CLONE_PARENT	informacje o rodzicu	współdzieli rodzica	staje się potomkiem
CLONE_THREAD	grupa wątków	potomek dołącza do grupy wątków rodzica	potomek tworzy nową grupę

Zarówno wywołanie systemowe `fork()`, jak i funkcja `pthread_create()` korzystają z wywołania systemowego `clone()` do tworzenia procesów i wątków.

Przykład wywołania tworzącego wątek:

```
int tid = clone(worker, stack_top,
    CLONE_VM | CLONE_SIGHAND | CLONE_THREAD, arg);
```

gdzie `stack_top` wskazuje na szczyt wcześniej zaalokowanego stosu nowego wątku.

Przykład wywołania tworzącego proces:

```
int pid = clone(child_fn, stack_top, SIGCHLD, arg);
```

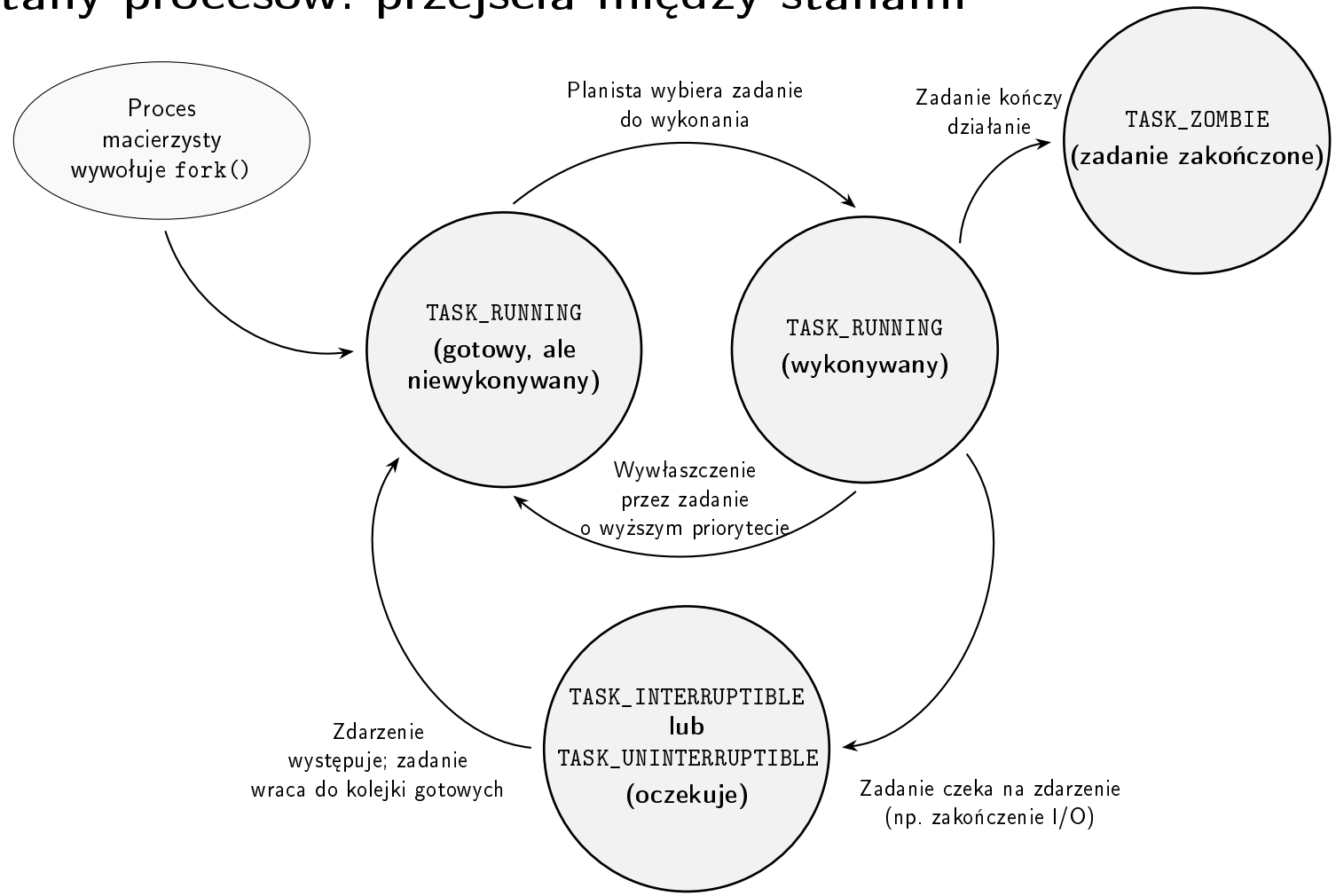
gdzie nie ustawiono flag `CLONE_*`, zaś `SIGCHLD` zajmuje najniższy bajt argumentu `flags` i określa sygnał wysyłany do procesu rodzica po zakończeniu procesu potomnego.

Stany procesów według jądra Linuksa

- `TASK_RUNNING` – proces albo się wykonuje, albo czeka na wykonanie w kolejce procesów gotowych
- `TASK_INTERRUPTIBLE` – proces jest wstrzymany do czasu zajścia określonego zdarzenia. Proces może zostać zbudzony przez przerwania sprzętowe, zwolnienie zasobu, na który proces czeka, albo dostarczenie sygnału
- `TASK_UNINTERRUPTIBLE` – wybudzenie może nastąpić po uzyskaniu zasobu, na który proces czekał. Sygnały nie wybudzają zadania lecz mogą pozostać oczekujące do czasu jego wybudzenia. [2]
- `TASK_ZOMBIE` – wykonanie procesu zostało zakończone, ale proces rodzica nie odebrał jeszcze informacji o jego zakończeniu (rodzic nie użył jeszcze wywołania systemowego `wait()` lub `waitpid()`)
- `TASK_STOPPED` – wykonanie procesu zostało zatrzymane (wskutek odebrania sygnału `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`)

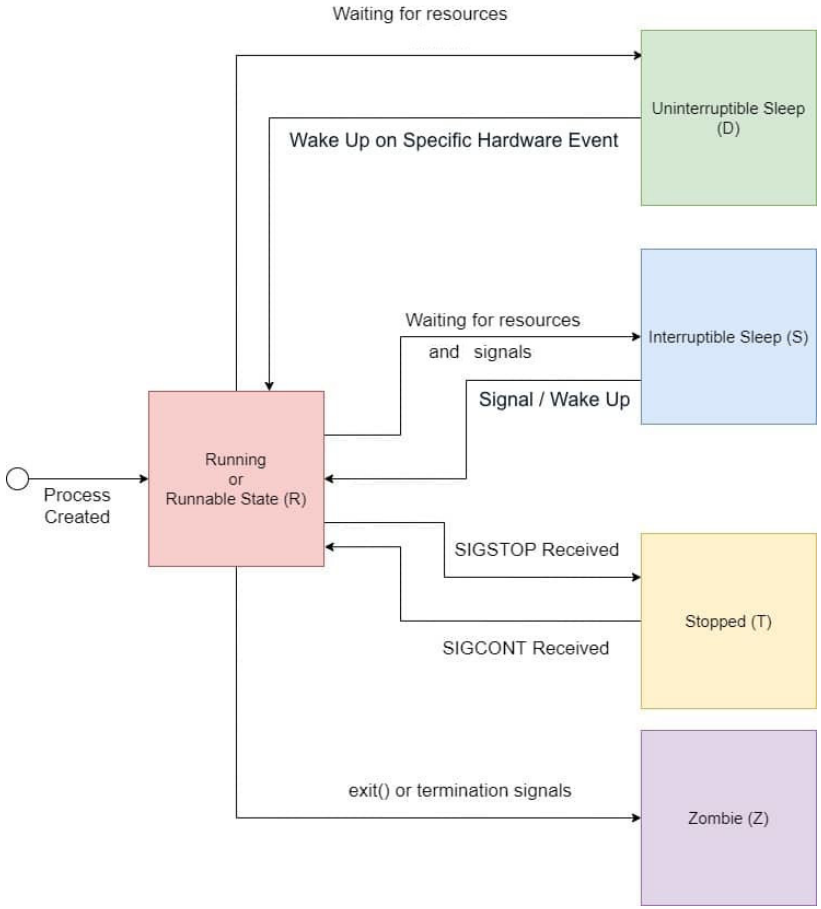
[2] TASK_KILLABLE, alternatywny stan `TASK_UNINTERRUPTIBLE` reagujący na krytyczne sygnały wprowadzony w łatce do jądra 2.6.25

Stany procesów: przejścia między stanami



http://learnlinuxconcepts.blogspot.com/2014/03/process-management.html

Stany procesów



Obserwowanie stanów: polecenie ps

```
$  ps a
    PID TTY      STAT   TIME COMMAND
   5270 tty2     SNsl+   0:00 /usr/libexec/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=pop /usr/bin/gnome-s
   5272 tty2     S<l+  59:28 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -nolisten tcp -bac
   5924 pts/0    SNs    0:37 /usr/bin/zsh
1680036 pts/7    TNs+   0:00 zsh -Z -g
1871214 pts/5    TN     0:00 pdflatex so05-procesy.tex
1934828 pts/5    SNLl+  0:36 gimp process-finite-state-machine.jpeg
1940687 pts/0    SN     0:00 /usr/bin/zsh
1940688 pts/0    RN+    0:00 ps a
```

Stany procesów wg `man ps`

R (*running or runnable*) wykonujący się lub gotowy do wykonania

S (*interruptible sleep*) śpiący, czeka na zdarzenie, można wybudzić sygnałem

D (*uninterruptable sleep*) – nieprzerywalny sen (zwykle I/O)

T zatrzymany przez sygnał sterujący (np. `SIGSTOP`)

t zatrzymany przez program śledzący (debugger)

Z (defunct) "zombie" – zakończony, ale nie zebrany przez proces rodzica

X (*dead*) – *should never be seen*

< proces o wysokim priorytecie (nie jest „uprzejmy")

N proces o niskim priorytecie

L (pages locked into memory) – proces ze stronami zablokowanymi w pamięci

s – leader sesji

l – proces wielowątkowy

\+ – proces pierwszego planu

Procesy i wątki: obserwowanie procesów wielowątkowych

```
$ ps -eo stat,pid,ppid,tgid,tid,lwp,nlwp,vsz,rsz,cmd
STAT     PID    PPID    TGID     TID     LWP NLWP    VSZ   RSZ CMD
...
SNl  1217879 1217875 1217879 1217879 1217879    4 321996 29420 /usr/bin/gnome-terminal.real --wait
...

$ ps  -m -o stat,pid,ppid,tgid,tid,lwp,nlwp,vsz,rsz,cmd -p 1217879
STAT     PID    PPID    TGID     TID     LWP NLWP    VSZ   RSZ CMD
-    1217879 1217875 1217879       -       -    4 321996 29420 /usr/bin/gnome-terminal.real --wait
SNl        -       -       - 1217879 1217879    -      -     - -
SNl        -       -       - 1217880 1217880    -      -     - -
SNl        -       -       - 1217881 1217881    -      -     - -
SNl        -       -       - 1217882 1217882    -      -     - -
```

Z dokumentacji `ps`:

`tgid` - identyfikator grupy wątków, do której należy zadanie (alias `pid`)

`tid` - unikalny numer reprezentujący jednostkę wykonywalną (*dispatchable entity*) (alias `lwp`)

`lwp` - ID lekkiego procesu (wątku) jednostki wykonywalnej (alias tid)

`rsz` - (*resident set size*), przybliżony rozmiar pamięci fizycznej procesu, która obecnie znajduje się w RAM-ie, podany w KiB.

`vsz` - rozmiar przestrzeni pamięci wirtualnej procesu w KiB, także mapowanie pamięci, których strony nie muszą obecnie znajdować się w RAM-ie

Procesy systemowe w Linuksie – wyłączne wątki jądra

Jądro nie jest procesem w tradycyjnym sensie, ale zarządcą procesów.

Wyłączne wątki jądra (`Kernel Threads`) wykonują zadania krytyczne dla działania systemu

- działają wyłącznie w trybie jądra (w przestrzeni adresowej jądra), nie mają przypisanej przestrzeni adresowej użytkownika, nie mogą wykonywać kodu użytkownika
- działają w kontekście procesu (posiadają unikatowy `PID`), podlegają planowaniu, mogą wykonywać operacje blokujące i być wywłaszczane
- nie komunikują się z użytkownikami (nie trzeba terminali)
- tworzone wyłącznie przez jądro (funkcja `kthread_create`) lub podczas startu systemu
- nie można ich zabić

Wątki jądra Linuksa

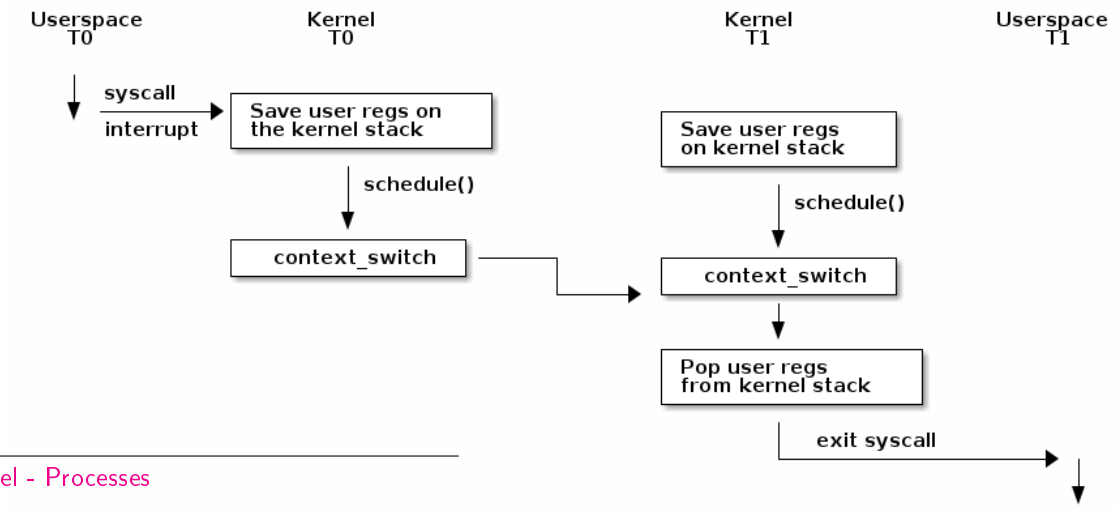
Informacje o procesie PID 2 (`[kthreadd]`) i jego wątkach potomnych:

```
$ ps l -p 2 --ppid 2
F   UID     PID    PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
1     0       2       0  20   0      0     0 -      S    ?          0:00 [kthreadd]
1     0       3       2  20   0      0     0 -      S    ?          0:00 [pool_workqueue_release]
1     0       4       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-rcu_g]
1     0       5       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-slub_]
1     0       6       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-netns]
1     0       8       2   0 -20      0     0 -      I<   ?          0:00 [kworker/0:0H-events_highpr
...
```

- wątki jądra mają numer PID (identyfikator procesu) tak jak zwykłe procesy użytkownika
- VSZ i RSS wynoszą 0, co oznacza, że wątki jądra nie zajmują pamięci użytkownika.
- wątki jądra nie mają przypisanego terminala (TTY)
- wątek jądra `[kthreadd]` ma PPID=0 i nie ma procesu nadrzędnego
- nawiasy kwadratowe, np. `[kthreadd]`, są typowym sposobem prezentowania wątków jądra

Przełączenie kontekstu zadania z trybu użytkownika

- przed zmianą kontekstu musi dojść do zmiany trybu pracy z trybu użytkownika na tryb jądra w wyniku wywołania systemowego lub przerwania (wejście T0 do jądra)
- rejestry użytkownika zostają zapisane na stosie jądra bieżącego zadania
- kod jądra (planista) wybiera zadanie do dalszego wykonania (`schedule()`)
- podczas przełączania kontekstu zapisywany jest kontekst jądra bieżącego zadania i odtwarzany zapisany stan innego zadania (`context_switch`), w tym jego stos
- wybrane zadanie wznawia wykonanie w jądrze, odtwarza swoje rejestry użytkownika ze stosu jądra i wraca do przestrzeni użytkownika



Wywłaszczanie procesu

Algorytm szeregujący zwany **planistą** lub **dyspozytorem** może wstrzymać aktualnie wykonywane zadanie (proces lub wątek), aby umożliwić działanie innemu zadaniu - zadanie zostaje **wywłaszczone**

Do wywłaszczenia może dojść, gdy:

- nowe zadanie o wyższym priorytecie staje się gotowe do wykonania
- zadanie wyczerpie swój kwant czasu
- po wywołaniu systemowym lub obsłudze przerwania przełączenie może nastąpić przed powrotem do przestrzeni użytkownika
- przerwanie może nadejść asynchronicznie podczas wykonywania kodu użytkownika; samo jego nadejście nie oznacza jednak, że zadanie zostanie wywłaszczone

Czy proces może być wywłaszczony, jeśli przebywa w trybie jądra?

Wywłaszczenie jądra

- jądra współczesnych systemów są **wielobieżne** (*reentrant*) i **wywłaszczalne** (*preemptive*)
- **Wielobieżność** (*reentrancy*) pozwala na równoległe wykonywanie kodu jądra na wielu procesorach
- **Wywłaszczalność** pozwala wstrzymać wykonywanie kodu jądra i uruchomić inne gotowe zadanie. Na jednym procesorze nie zapewnia równoległości, ale może skrócić czas reakcji na pilniejsze zadanie
- przełączenie następuje w bezpiecznym punkcie: w sekcjach krytycznych preempcja jest wyłączona lub odraczana.
- przerwanie sprzętowe może wywołać decyzję planisty o wywłaszczeniu aktualnie wykonywanego zadania, nawet wykonywanego w trybie jądra, skracając czas reakcji na zdarzenie urządzenia
- w Linuksie zakres preempcji zależy od konfiguracji jądra: możliwy jest brak wymuszonej preempcji, preempcja dobrowolna lub pełna

Pytania sprawdzające

- Wymień trzy podstawowe stany procesu w systemie operacyjnym i wskaż, w jakich sytuacjach może nastąpić przejście między nimi
- Co to jest wywłaszczenie procesu i kiedy może do niego dojść?
- Jaka jest różnica między wątkami poziomu użytkownika i wątkami poziomu jądra?