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
- wykonanie procesu musi przebiegać w sposób sekwencyjny

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
- monitorowanie działania procesów i dostarczanie narzędzi do diagnostyki
- zarządzanie przydziałem pamięci dla procesów, wirtualizacja pamięci i ochrona dostepu

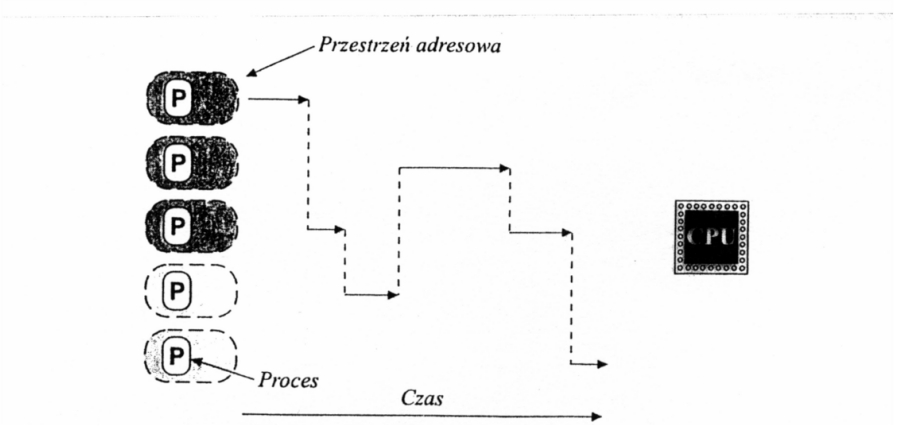
Procesy: rodzaje

Proces stanowi jednostkę pracy w systemie.

System składa się ze zbioru procesów:

- procesy systemu operacyjnego (wykonują kod systemu)
- procesy użytkowników (wykonują kod programów użytkowników)
- procesy działające w tle (demony) - wykonują zadania bez bezpośredniej interakcji z użytkownikiem (usługi)
- procesy interaktywne (np. terminale, aplikacje GUI)
- procesy wsadowe (batch) - uruchamiane bez interakcji użytkownika, często jako zadania wsadowe wykonywane sekwencyjnie

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



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

Przetwarzanie równoległe - jednoczesne wykonywanie wielu procesów na różnych procesorach, procesy są rzeczywiście wykonywane jednocześnie.

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:

- normalne wyjście po zakończeniu pracy (dobrowolne)
  - Unix/Linux: `exit()`
  - Windows WinAPI: `ExitProcess()`
- wyjście zakończone błędem (dobrowolne), gdy proces wykryje sytuację uniemożliwiającą dalsze wykonie
- krytyczny błąd (niedobrowolne), w wyniku wykonania niepoprawnej instrukcji
- zabicie przez inny (uprawniony) proces (niedobrowolne), poprzez wywołanie funkcji systemowej
  - Unix/Linux: `kill()`
  - Windows WinAPI: `TerminateProcess()`

Hierarchia procesów

- więź między procesami **potomnymi** (*child*) i **nadrzędnymi** (*parent*) tworzy hierarchię (np. Linux), strukturę drzewiastą.
  Pierwszy proces (korzeń) startuje podczas inicjacji systemu, np. `init`, `systemd`
- każdy proces posiada tylko jednego rodzica
- proces wraz ze wszystkimi potomkami i przodkami tworzy **grupę procesów**, które wspólnie moga odbierać sygnały, choć każdy z nich może dostarczać indywidualną funkcję obsługi sygnału lub go zignorować
- nie wszystkie systemy stosują hierarchę.
  W Windows procesy nie tworzą wyraźnej hierarchii. Proces nadrzędny uzyskuje **uchwyt** do potomka, za pomocą którego może go kontrolować ale uchwyt może być przekazany innemu procesowi

Hierarchia procesów

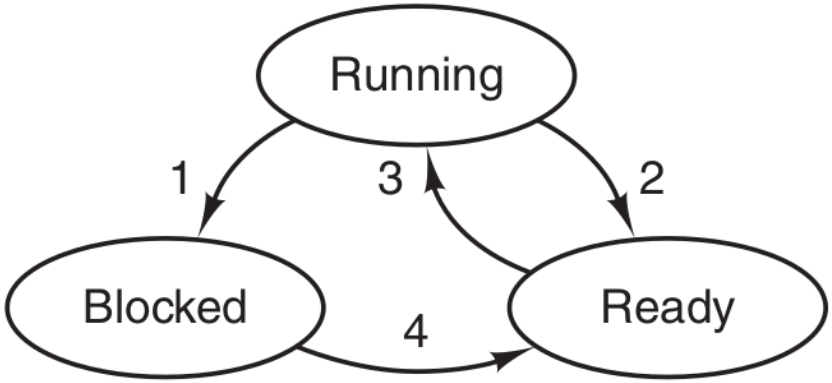
```
$ pstree
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-acpid
        |-atop
        |-atopacctd
        |-avahi-daemon---avahi-daemon
        |-blueman-mechani---2*[{blueman-mechani}]
        |-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** – proces czeka na przydział procesora
- **bieżący** (aktywny) – aktualnie używający CPU, są wykonywane instrukcje
- **oczekujący** (zablokowany) – czeka na wystąpienie jakiegoś zdarzenia (np. zakończenia operacji wejścia-wyjścia)

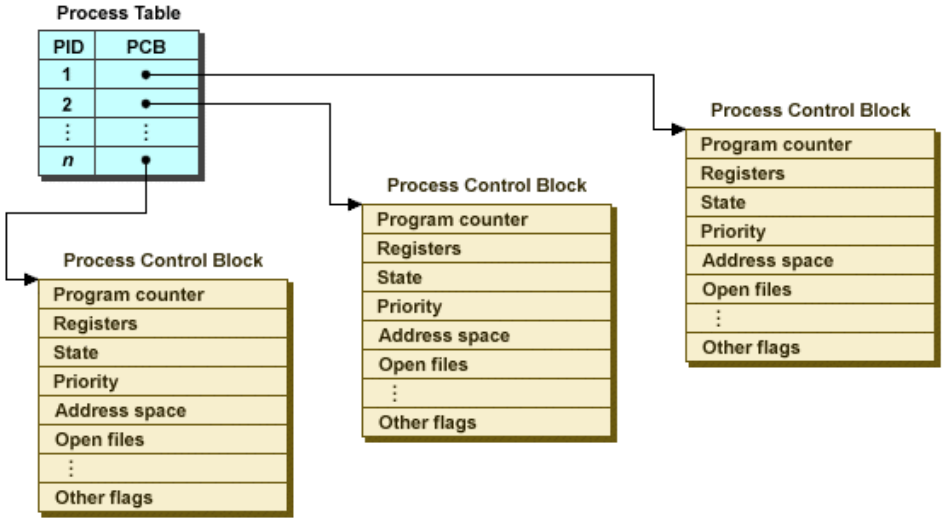
Procesy: zmiana stanu



1. proces blokowany w oczekiwaniu zdarzenie (oczekiwanie na wejście)
2. planista wybiera inny proces do wykonania
3. planista wybiera ten proces do wykonania
4. zachodzi zdarzenie odblokowujące proces (wejście staje się dostępne)

Procesy: zarządzanie

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



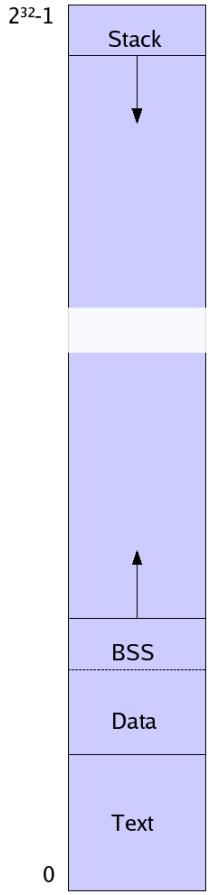
Blok kontrolny procesu zawiera wszystkie niezbędne informacje wymagane do zarządzania procesem, wszystko co musi być zapisane, gdy proces opuszcza stan bieżący, tak aby mozliwe było wznowienie jego wykonania

- stan procesu
- numer procesu
- licznik rozkazów, stosu
- rejestry
- ograniczenia pamięci
- wykaz otwartych plików
- informacja o planowaniu przydziału procesora
- informacja o wykorzystanych zasobach (rozliczanie)

Typowe pola bloku kontrolnego procesu

zarządzanie procesem	zarządzanie pamięcią	zarządzanie plikami
rejestry	wskaznik do informacji segmentu tekstu	katalog głowny
licznik programu	wskaznik do informacji segmentu danych	katalog roboczy
słowo stanu programu	wskaznik do informacji segmentu stosu	deskryptowy plików
wskaznik 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 określona wirtualna przestrzeń adresowa, segment programu (*text*), danych zainicjowanych (*data*), danych niezainicjowanych (*bss*), sterty (*stack*), stosu (*heap*).

```
$ size /bin/ps
   text    data     bss     dec     hex filename
  71928     769  131967  204664   31f78 /bin/ps
```

Plik: /proc/PID/maps

$ pmap PID

Przykład: polecenie ps

```
# 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
```

Wątki

Wątek – (lekki) proces działający w tej samej wirtualnej przestrzeni adresowej, co tworzący go (ciężki) proces. Stan wątku jest zdefiniowany przez małą, odrębną ilość danych (własny stan rejestrów i stos).

Grupa równoprawnych wątków współdzieli zasoby procesu

- kod
- przestrzeń adresową
- otwarte pliki
- zasoby systemu
- należy do tego samego użytkownika

Wątki ze sobą współpracują, a nie współzawodniczą (tak jak procesy).

Wątek, to podstawowa jednostka wykorzystania procesora.

Procesy i wątki

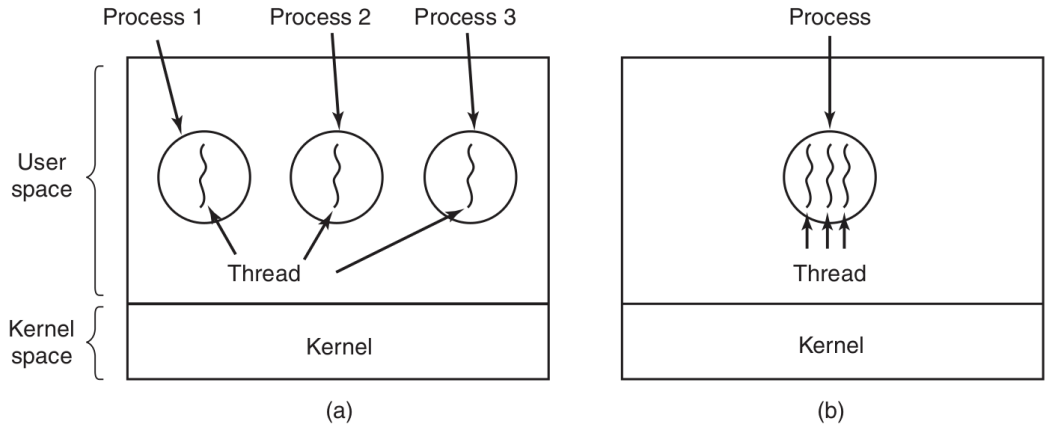
Procesy grupują zasoby, zaś wątki są jednostkami wykonawczymi pracującymi na CPU.

Wątki posiadają własny:

- licznik programu, wskazujący następną instrukcję do wykonania
- stos z historię wywołań procedur
- rejestry, przechowujące wartości zmiennych
- stan, identycznie do stanu procesu: gotowy, bieżący, zablokowany, zakończony

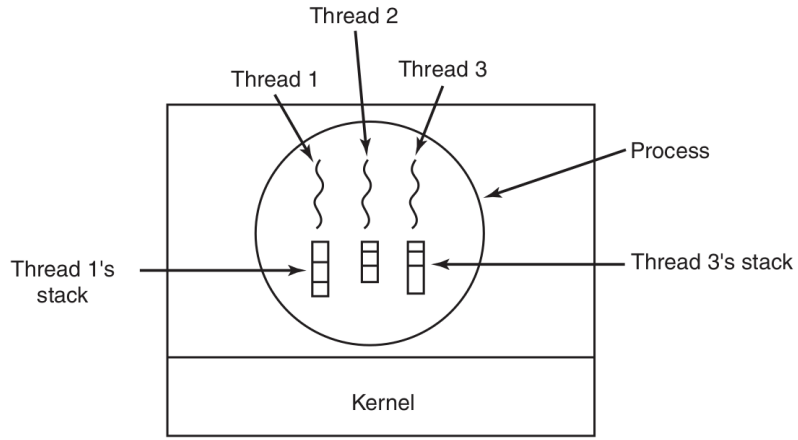
Wielowątkowość - gdy możliwe jest występowanie wielu wątków w procesie

Procesy i wątki



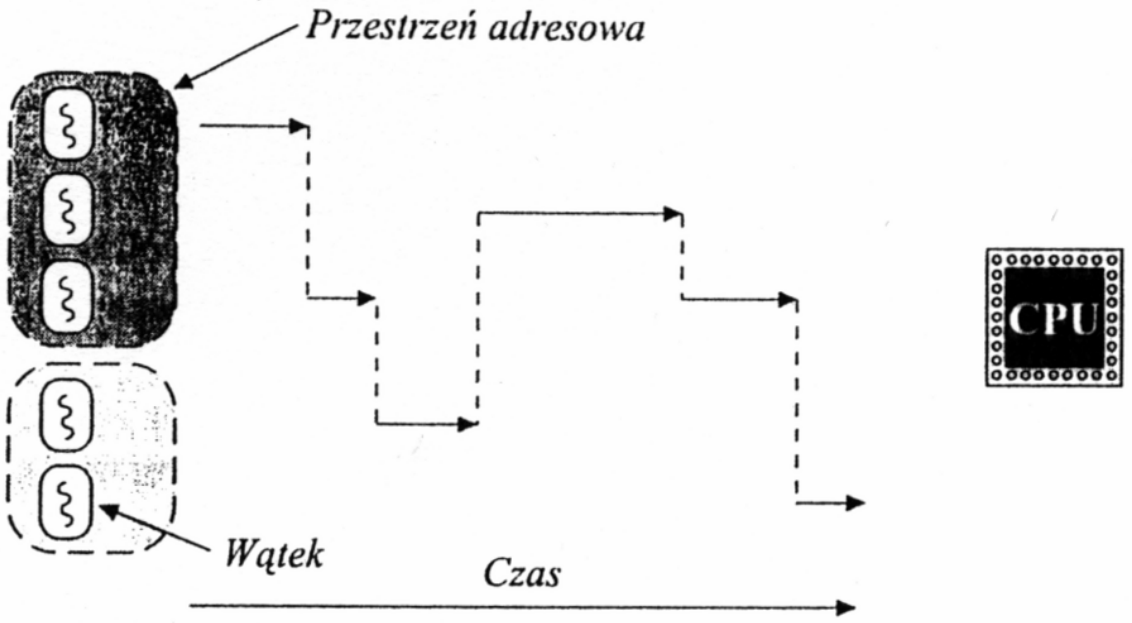
Proces	Wątek
przestrzeń adresowa	licznik programu
zmienne globalne	rejestry
otwarte pliki	stos
procesy potomne	stan
zaległe alarmy	
sygnały i procedury obsługi sygnałów	
informacje dotyczące statystyk	

Stos procesu



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

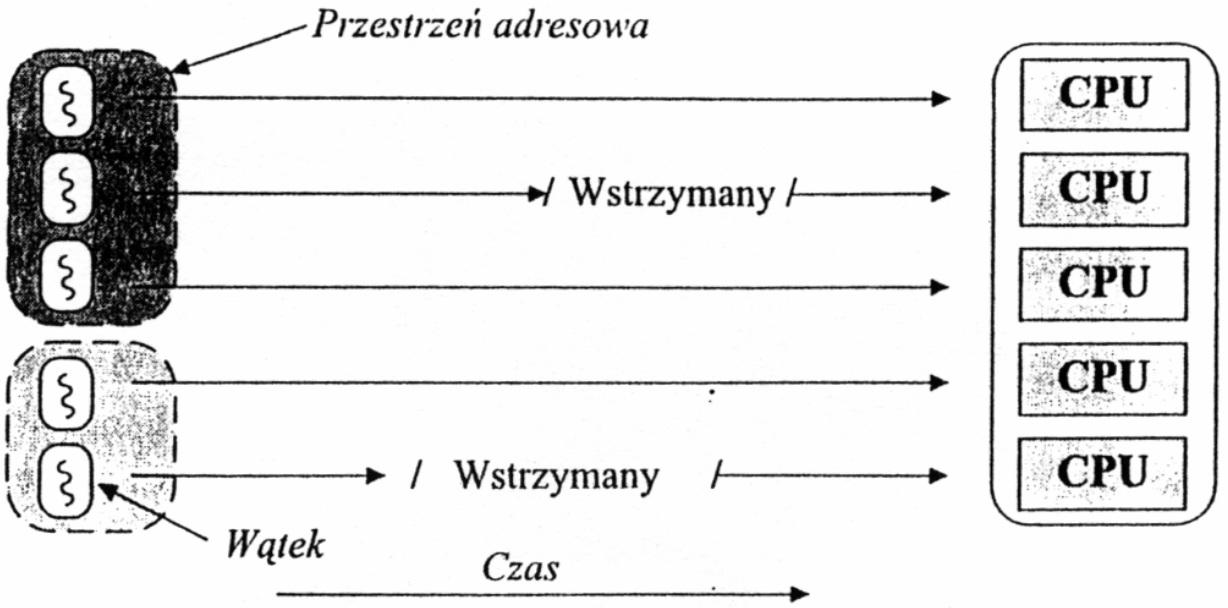
Procesy wielowątkowe w systemie z jednym CPU



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

Niektóre architektury CPU posiadają wsparcie sprzętowe wielowątkowości pozwalając na przełaczenie miedzy wątkami w skali nanosekund (*Hyper-Threading*)

Procesy wielowątkowe w wieloprocesorze

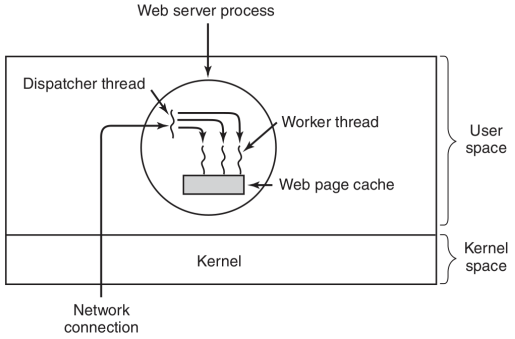


Rysunek 3-3. Procesy wielowątkowe w wieloprocesorze

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

Zalety wątków

- przełączanie procesora między wątkami jest łatwiejsze (szybsze) niż między zwykłymi (ciężkimi) procesami
- tworzenie i niszczenie wątków prostsze i szybsze niż procesów (nawet 10-100 razy szybsze)
- lepsze wykorzystanie zasobów systemu komputerowego
- lepsza realizacja przetwarzania współbieżnego na maszynach o pamięci współdzielonej (SMP)




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

(a)

```
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);
}
```

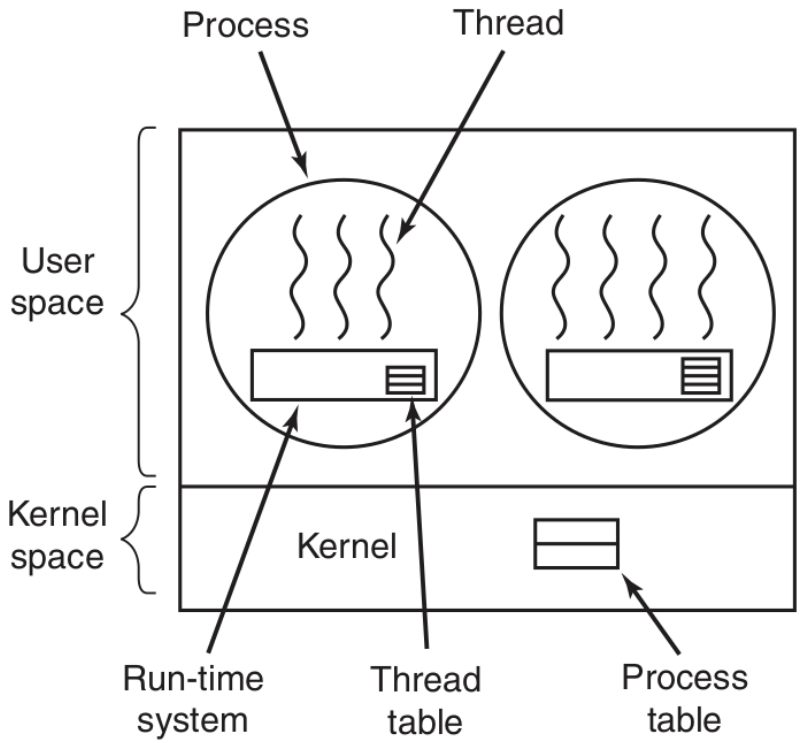
(b)

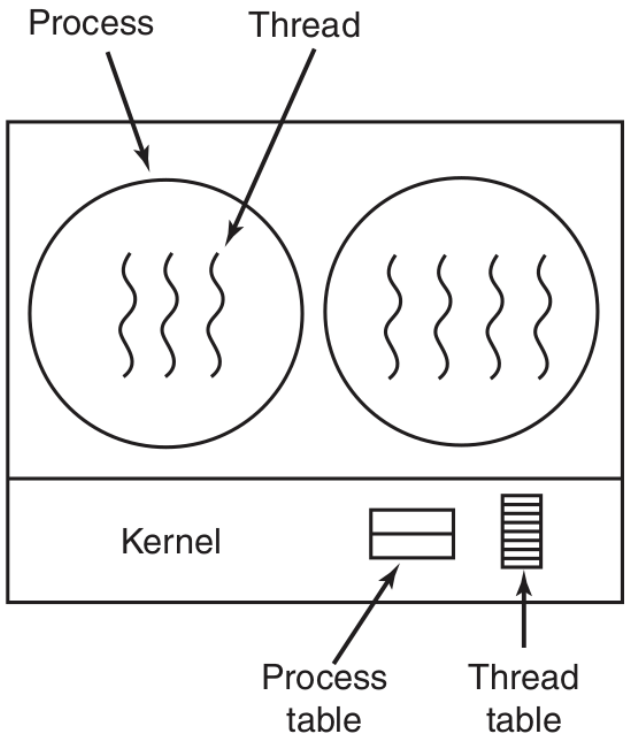
Przykład serwera realizowanego za pomocą wątku dyspozytora i wątków roboczych

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_yeld`) tak aby dać innym wątkom szansę na wykonanie

Wątki w przestrzeni użytkownika i w przestrzeni jądra





Wątki użytkownika (*user threads*)

- zarządzane przez biblioteki użytkownika, a nie bezpośrednio przez jądro systemu operacyjnego
- moga być przezroczyste dla jądra, implementacja nie wymaga wsparcia wielowątkowości w systemie
- przykłady: p-wątki (*Pthreads*) wątki wg normy POSIX

Wątki jądra (*kernel threads*)

- zarządzane bezpośrednio przez jądro systemu operacyjnego
- przykłady: *Linux Kernel Threads*, *Windows Threads*

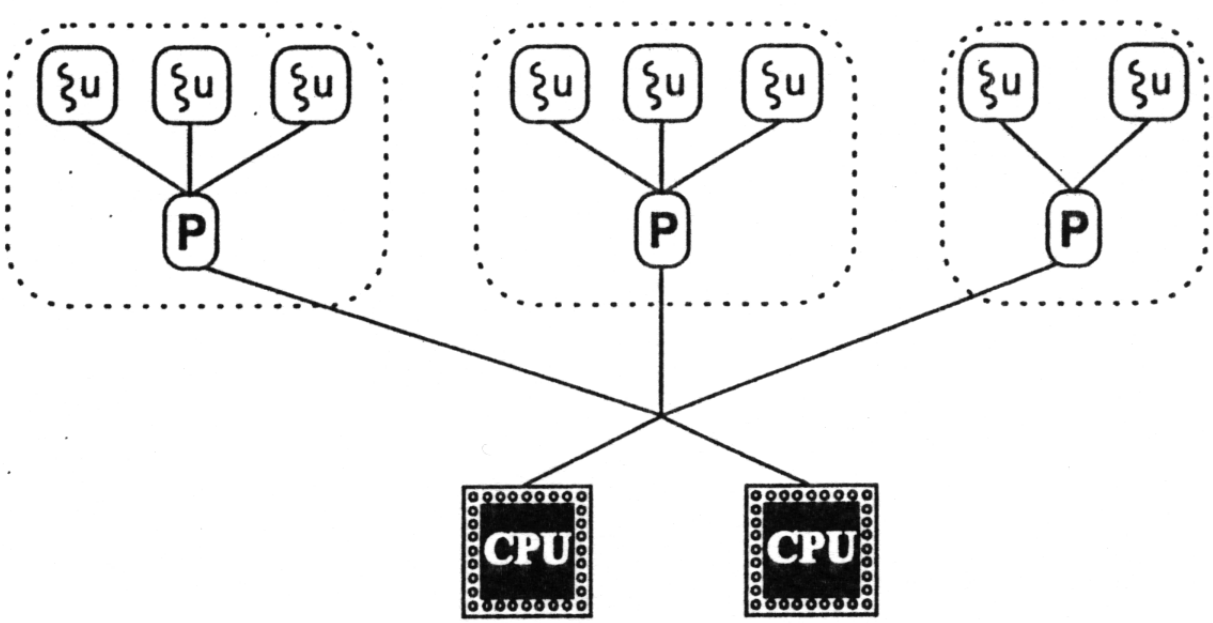
Wątki hybrydowe (*hybrid threads*)

- połączenie wątków użytkownika i jądra, korzystają z mechanizmów wątków użytkownika i jądra, aby zbalansować zalety obu podejść
- procesy lekkie (*lightweight processes LWP*) - działają w przestrzeni użytkownika na pojedynczym wątku jądra, dzielą swoją przestrzeń adresową i zasoby systemowe z innymi LWP w ramach tego samego procesu

Wątki użytkownika

- wątki działające w przestrzeni użytkownika, implementowane przez bibliotekę
- każdy proces przechowuje własną **tablicę wątków**, wskazującą na zasoby każdego wątku (licznik programu, rejestry, stos)
- kontekst wątku zapamiętywany i odtwarzany jest z poziomu użytkownika
- jądro zarządza procesem (nie wie o istnieniu wątków użytkownika), wywłaszczając proces użytkownika wywłaszcza związane z nim wątki
- wątki użytkownika są wydajne, nie zużywają zasobów jądra (jeśli nie są związane z procesem lekkim)
- przełaczne wątków jest bardzo szybkie w stosunku do wywołań systemowych
- każdy proces może mieć indywidualny schemat zarządzania wątkami

Wątki w przestrzeni użytkownika

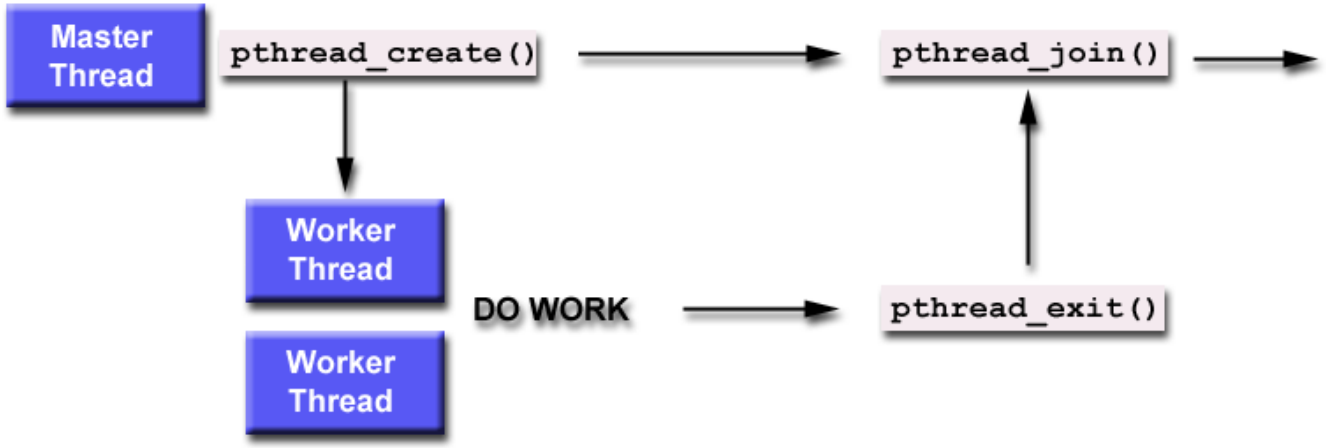


Problemy wątków użytkownika

- biblioteka szereguje wątki, jądro procesy; wątki zwiększają poziom współbieżności, ale nie równoległości
- blokujące operacje (wywołania systemowe, błedy strony) powodują zablokowanie procesu, więc i wszystkich wątków
- wywołanie systemowe generują przerwanie i przejście do trybu jądra, jest to kosztowne
- uruchomiony wątek zajmuje CPU nie dając możliwości działania innym wątkom, dopóki nie zrzeknie się procesora (*yeld*)

p-wątki (pthreads)

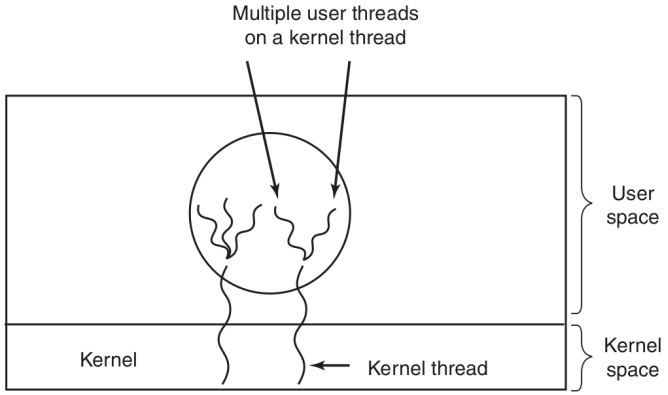
Biblioteka *pthreads* (zgodna ze standardem POSIX) udostępnia abstrakcję wątków całkowicie na poziomie użytkownika, tworzenie, usuwanie, synchronizowanie, szeregowanie oraz zarządzanie wątkami bez udziału jądra



Wątki jądra

- jądro zarządza wątkami, tabela wątków znajduje się w przestrzeni jądra a nie procesu
- tworzenie oraz niszczenie wątków odbywa się przez odwołania do jądra, jest to kosztowniejsze niż w przypadku wątków użytkownika
- wszystkie wywołania blokujące są zaimplementowane jako funkcje systemowe
- w momencie zablokowania wątku system może wznowić wykonywanie innego, gotowego wątku (także z innego procesu)
- wątki jądra nie wymagają implementacji nieblokujących wywołań systemowych

Hybrydowe implementacje wątków



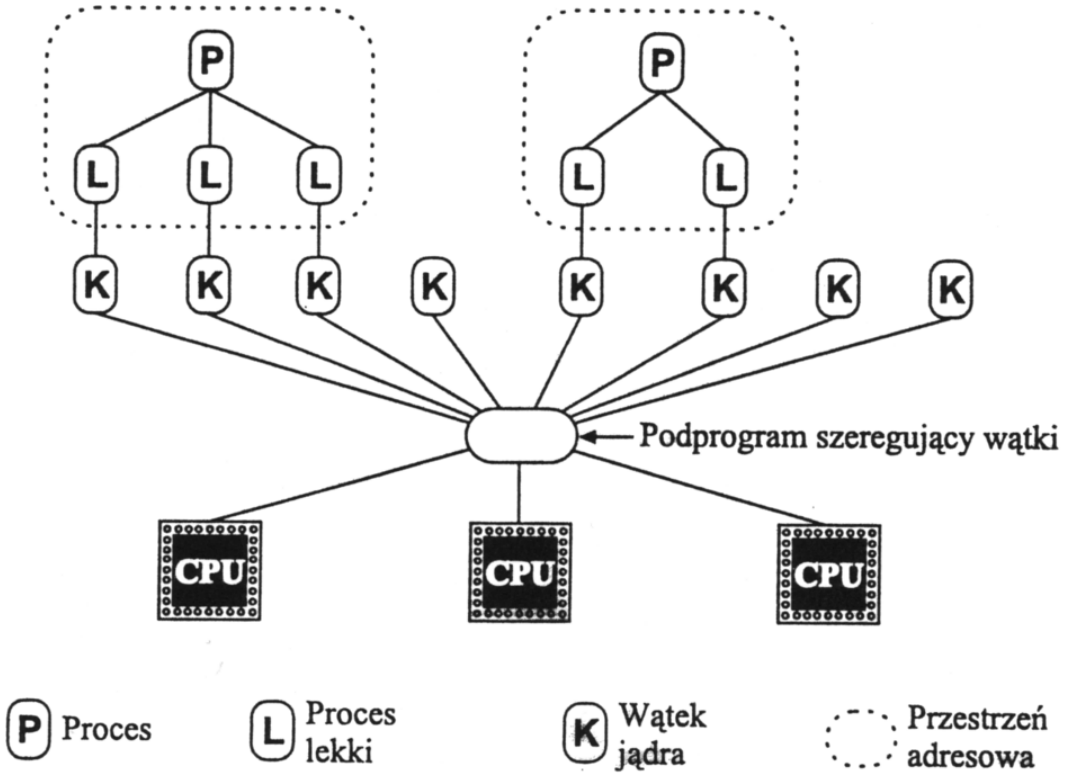
- próba połaczenia zalet zarządzania wątkami na poziomie użytkownika i na poziomie jądra
- wątki na poziomie jądra są zwielokrotniane na poziomie użytkownika
- programista może wybrać, ile wątków jądra wykorzystać i na ile wątków użytkownika je zwielokrotnić
- jądro systemu zarządza wyłącznie wątkami jądra
- zarządzanie wątkami użytkownika odbywa w ramach procesu, analogicznie do tradycyjnego modelu wątku użytkownika

Procesy lekkie, LWP

- Proces lekki (*light-weight process*, LWP) jest wspieranym przez jądro wątkiem z przestrzeni użytkownika.
- System udostępniając procesy lekkie musi także udostępniać wątki jądra. W każdym procesie może być kilka procesów lekkich, z których każdy jest wspierany przez osobny wątek jądra.
- Procesy lekkie są niezależnie szeregowane, współdzielą przestrzeń adresową, mogą wywoływać funkcje systemowe, które powodują wstrzymanie w oczekiwaniu na wejście-wyjście lub zasób. Mogą się wykonywać na różnych procesorach.
- Operowanie na procesach lekkich jest kosztowne, gdyż wymaga użycia wywołań systemowych (przełączeń trybu). Trzeba zapewnić synchronizację w dostępie do współdzielonych danych.

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

Procesy, procesy lekkie i wątki



Procesy i wątki: porównanie

- **Null fork:** czas mierzony od utworzenia poprzez jego szeregowanie, wykonywanie, aż do zakończenia procesu (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 zawierającą warunek (miara obciążenia spowodowanego wzajemną synchronizacją)

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

	UT	KT	P
Null fork	34	948	11300
Signal-wait	37	441	1840

[1] VAX, system uniksopodobny

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.); kopiowanie przy zapisie *copy-on-write* odkłada tworzenie rzeczywistej kopii, do momentu gdy fragment pamięci zostanie zmieniony
- zwiększa plikom związanym z tym procesem liczniki w tablicy plików i i-węzłów
- przekazuje identyfikator potomka procesowi macierzystemu i wartość zero procesowi potomnemu

Tworzenie procesu w UNIX/Linux

Wywołanie systemowe `exec()` umieszcza w obszarze pamięci 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 `exec()` realizowane przez funkcję `execve()` lub odmiany tej funkcji

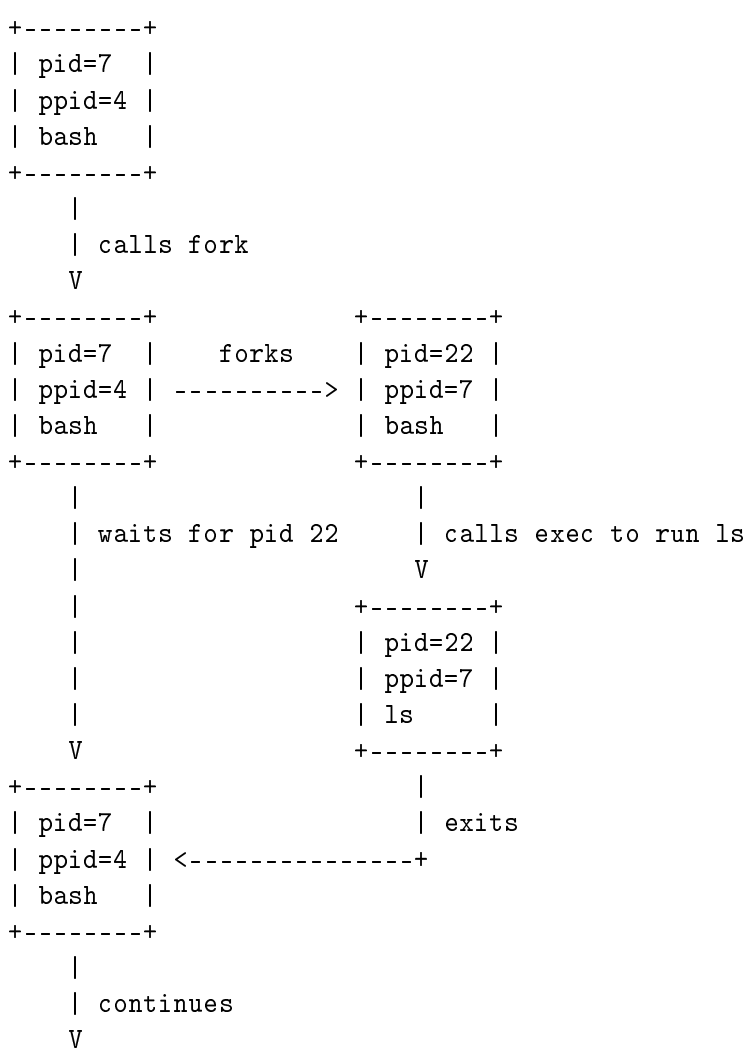
Procesy: tworzenie

Implementacja prostej powłoki

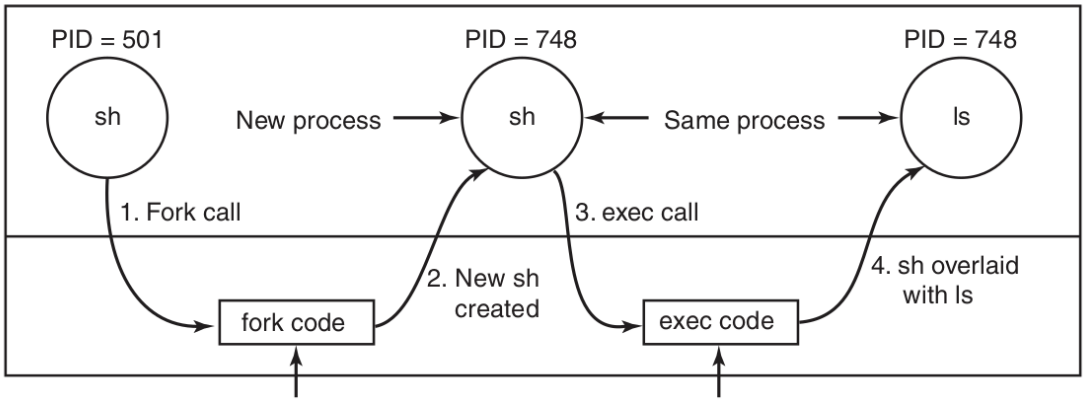
```
while(TRUE) {
   type_command();
   read_command(command, params);

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

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



Uruchomienie procesu w powłoce



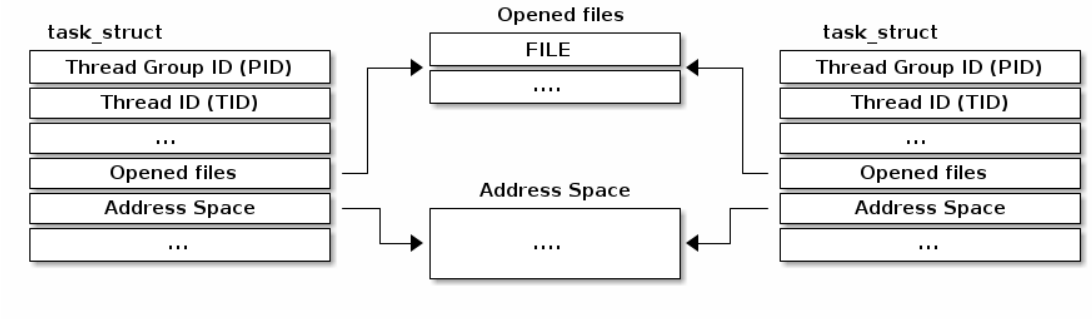
1. przydziela pamięć dla struktury procesu potomnego
2. wypełnia strukturę zadania potomnego danymi rodzica
3. przydziela pamięć dla stosu i przestrzeni użytkownika procesu potomnego
4. wypełnia przestrzeń użytkownika procesu potomnego danymi procesu rodzica
5. przydziela procesowi potomnemu PID
6. ustawia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

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

Procesy: zarządzanie w systemie GNU/Linux

W jądrze linuksa procesy (i wątki) reprezentowane są w postaci **zadań** (*task*) przez strukturę danych (`task_struct`)

- struktura zadań reprezentuje kontekst wykonawczy
- zasoby zadania nie są osadzone w strukrurze ale struktura posiada wskaźniki do zasobów
- jednowątkowy proces użytkownika to pojedyncze zadanie, wielowątkowy proces to wiele zadań (po jednym na wątek) ale dzielących wspólne zasoby



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), który pozwala identyfikować zadania powiązane procesem
- przestrzeń adresowa (*address space*): definiuje przestrzeń wirtualną zadania
- informacje o przydziale procesora (*scheduling info*): status zadania, politykę przydziału procesora, parametry tej polityki (priorytet dynamiczny), flaga *need_resched*
- informacja o programie wykonywalnym (*executable info*): identyfikuje plik, który jest obrazem wykonywanego zadania
- informacja o sygnałach (*signal info*): zawiera 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*, czas wykonywania się zadania, wykorzystywaną pamięć, liczbę 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*): tablica plików otwartych przez zadanie
- 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	przestrzen adresowa	powstaje nowy wątek	powstaje nowy proces
CLONE_FS	infomracje o systemie plików katalog roboczy, główny, umask	wspólny katalog roboczy	niezależny katalog roboczy
CLONE_FILES	tablica otwartych plików	współdzielenie deskryptorów	koipie deskryptoów
CLONE_SIGHAND	tablica obsługi sygnałów	współdzieli	kopiuje tablicę
CLONE_PARENT	informacje o rodzicu	nowy wątek współdzieli rodzica	nowy wątek staje się potomkiem
CLONE_PID	PID	wątek ma ten sam PID	proces uzyskuje nowy PID

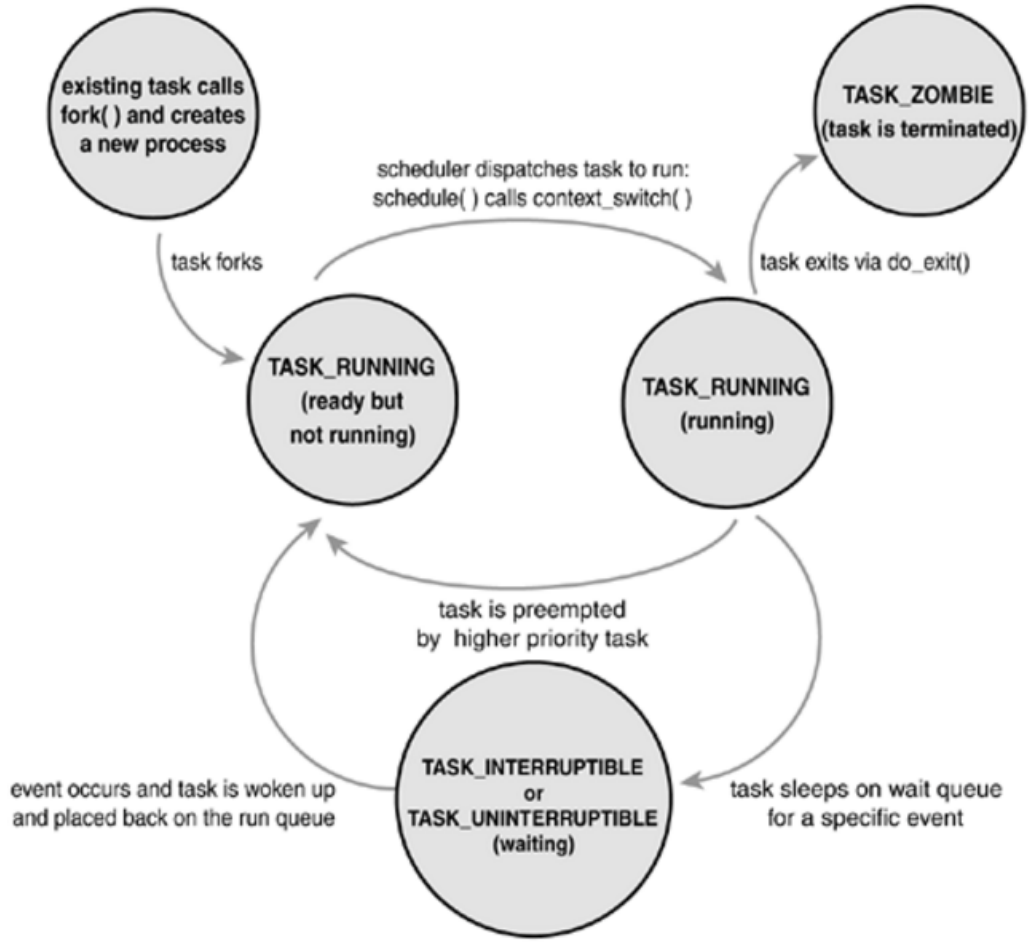
Zarówno wywołanie systemowe `fork()`, jak i funkcja `pthread_create()` korzystają z implementacji clone()

Stany procesów wg *Linux kernel*

- `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 jakiegoś zasobu systemowego, na który proces czeka, albo dostarczenie sygnału.
- `TASK_UNINTERRUPTIBLE` – wybudzenie może nastąpić po uzyskaniu zasobu, na który proces czekał. Sygnały są ignorowane [2]
- `TASK_ZOMBIE` – wykonanie procesu zostało zakończone, ale proces rodzica nie użył jeszcze wywołania systemowego `wait()` lub `waitpid()`, które zwraca informację o przerwanym procesie
- `TASK_STOPPED` – wykonanie procesu zostało zatrzymane (wskutek odebrania sygnału `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`.

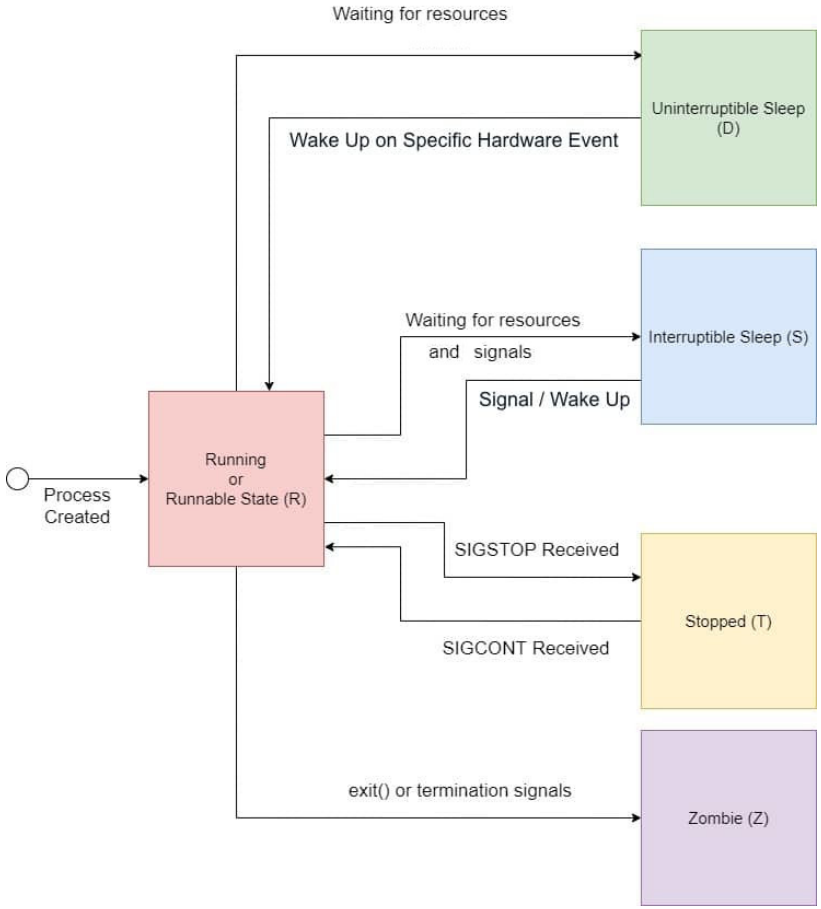
[2] TASK_KILLABLE, alternatywny stan `TASK_UNINTERRUPTIBLE` reagujący na krytyczne sygnały wprowadzony w łatce do jądrza 2.6.25

Stany procesów



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 zdolny do pracy (oczekujący w kolejce)

S (interruptible sleep) – śpiący (blocked)

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

T – zatrzymany przez sygnał sterujący

t – zatrzymany przez program śledzący (debugger)

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

X (dead) – (should never be seen)

< (high priority process) – proces o wysokim priorytecie

N (low priority process) – proces o niskim priorytecie

L (pages locked into memory) – proces ze stronami uwięzionymi 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) jednoski wykonywalnej (alias tid)

`rsz` - rozmiar zestawu rezydentnego (*resident set size*), czyli nieswapowana pamięć fizyczna, którą zadanie wykorzystało w kilobajtach (alias rsz)

`vsz` - rozmiar pamięci wirtualnej procesu w KiB

Procesy systemowe w Linux - wyłączne wątki jądra

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

Wątki jądra wykonujące zadania krytyczne dla działania systemu

- działają w kontekście procesu, moga wykonywać operacje blokujące
- działają wyłącznie w trybie jądra (w przestrzeni adresowej jądra)
- nie komunikują się z użytkownikami (nie trzeba terminali)
- tworzone są w chwili startu systemu i działają do czasu wyłączenia systemu

```
$ 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_highpri]
1     0      11       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-mm_pe]
1     0      12       2  20   0      0     0 -      I    ?          0:00 [rcu_tasks_kthread]
1     0      13       2  20   0      0     0 -      I    ?          0:00 [rcu_tasks_rude_kthread]
...
```

Blokowanie wątku:

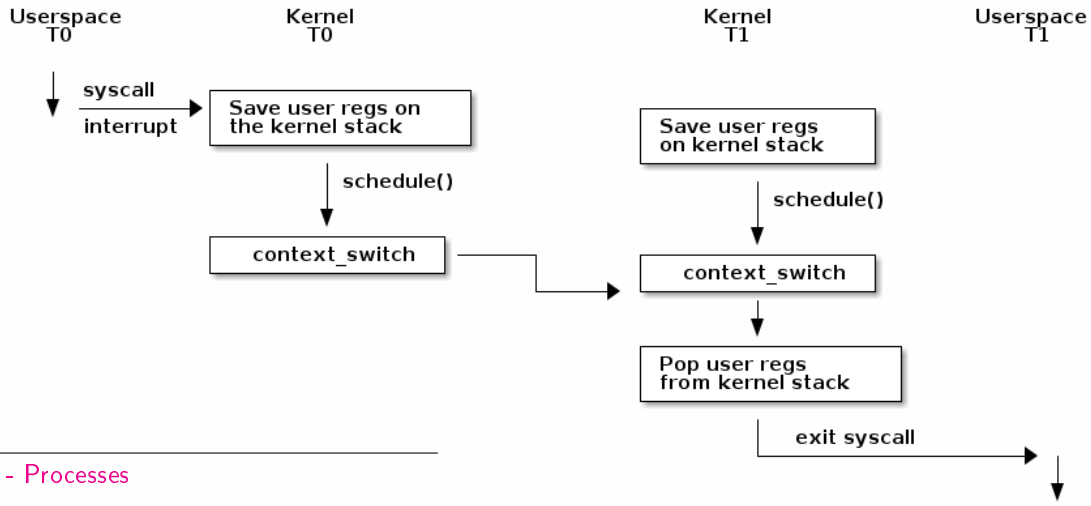
- ustaw bieżący stan wątku na TASK_UINTERRUPTIBLE lub TASK_INTERRUPTIBLE
- dodaj zadanie do kolejki zadań oczekujących
- wywołaj planistę, który pobierze nowe zadanie z kolejki zadań gotowych
- wykonaj przełączenie kontekstu na nowe zadanie

Budzenie zadania

- wybierz zadanie z kolejki zadań oczekujących
- ustaw stan zadania na TASK_RUNNING
- wstaw zadanie do kolejki zadań gotowych
- w systemie SMP, każdy procesor ma własną kolejkę, kolejki muszą być zrównoważone a procesory powiadomione sygnałami

Przełączenie kontekstu

- przed zmianą kontekstu musi dojść do zmiany trybu pracy na tryb jądra
- przejście do trybu jądra zachodzi w wyniku przerwania
- rejestry przestrzeni użytkownika są zapisywane na stosie jądra wskazywanym przez strukturę zadania
- w tym momencie może dojść do przełączenia kotekstu z innym wątkiem jądra (np. ponieważ bieżący wątek jest zablokowany lub wygasł przydzielony mu przedział czasu)



Wywłaszczanie procesu

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

Do wywłaszczenia może dojść gdy:

- proces wchodzi w stan `TASK_RUNNING` i jego priorytet jest wyższy od priorytetu procesu właśnie zawłaszczającego procesor
- kiedy proces wyczerpie swój kwant czasu
- wywłaszczenie procesu użytkownika może nastąpić przy powrocie do przestrzeni użytkownika z wywołania systemowego lub z procedury obsługi przerwania
- kiedy jednostka centralna wykonuje kod w trybie użytkownika

Wywłaszczenie jądra

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

- jądra uniksowe są **wielobieżne** (wielowejściowe, *reentrant*): kilka procesów może się wykonywać w trybie jądra w tym samym czasie
- w systemie jednoprocesorowym tylko jeden proces może działać, inne mogą czekać na CPU lub na zakończenie operacji wej/wyj (będąc zablokowanymi w trybie jądra)
- wystąpienie przerwania sprzętowego pozwala jądru wielobieżnemu na zatrzymanie procesu, nawet jeśli znajduje się on w trybie jądra. Wpływa to na zwiększenie szybkości obsługi urządzeń zewnętrznych.
- Linux (z jądrem w wersji $\leqslant$ 2.4) jest systemem operacyjnym z wywłaszczaniem procesów, ale bez wywłaszczania jądra. Jądro $\geqslant$2.6 jest już wielowejściowe, umozliwia wywłaszczenie zadania działajacego w trybie jądra i wykonującego wywołanie systemowe