Wątki

Procesy stanowią podstawową jednostkę, z której tradycyjnie składa się większość systemów operacyjnych. Procesy posiadają dwie podstawowe cechy:

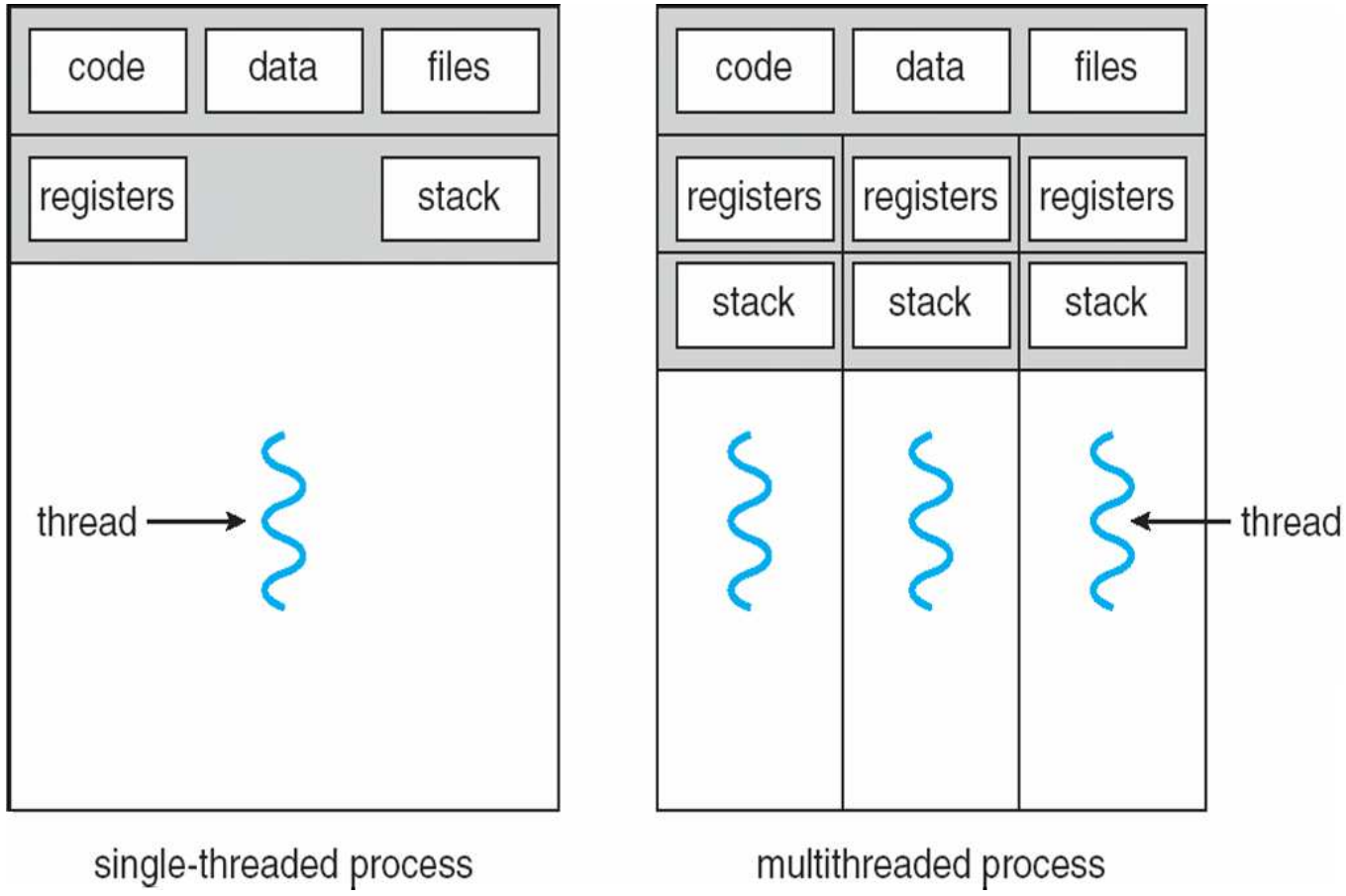
- kod programu załadowany do pamięci operacyjnej, wraz ze środowiskiem (zbiorem zmiennych posiadających wartości) i zasobami (np. zbiorem otwartych plików, i innymi przydzielonymi zasobami, np. portami komunikacyjnymi, itd.) przechowywanym przez jądro systemu,

- ślad wykonania (*execution trace*), tzn. ciąg wykonywanych instrukcji, wraz z niezbędnymi przy ich wykonywaniu konstrukcjami, np. licznikiem programu (PC), zbiorem rejestrów z zawartością, stosem wywoływanych procedur wraz z ich argumentami, itp.

W starszych systemach operacyjnych te dwie cechy procesu były integralnie ze sobą związane. Nowsze systemy pozostawiają pierwszą cechę procesom, ale drugą im odbierają na rzecz wątków (*threads*).

Procesy wielowątkowe

Wątki z natury rzeczy są jednostkami podrzędnymi procesów, tzn. jeden proces może składać się z jednego lub więcej wątków.



single-threaded process

multithreaded process

Wspólne dane wątków

Ponieważ wątki danego procesu współdzielą między sobą jego kod, zatem wszystkie zmienne globalne są ich wspólną własnością. Każdy z wątków może w dowolnym momencie odczytać lub nadpisać wartość zmiennej globalnej.

Jest to zarazem zaleta i wada. Zaleta polega na tym, że jest to najszybsza możliwa metoda komunikacji między wątkami, ponieważ wartości generowane przez wątek źródłowy są od razu dostępne w wątku docelowym. Jednak taki wspólny dostęp do danych wymaga **synchronizacji**, ponieważ wątek docelowy nie wie kiedy dokładnie dane zostały już wygenerowane i kiedy może je odczytać (wyobraźmy sobie, że dane te mają znaczną objętość, np. gdy jest to tablica lub struktura). Nie wiedząc tego, może odczytać stare dane, albo np. dane częściowo zmodyfikowane, które mogą w ogóle nie mieć sensu.

Podobnie synchronizacji wymaga sytuacja, gdy dwa wątki (lub więcej) chcą modyfikować dane. Bez synchronizacji takie operacje mogłyby doprowadzić do trwałego zapisania danych niespójnych.

Synchronizacji nie wymagają operacje odczytu wspólnych danych.

Prywatne dane wątków

Wywoływane przez wątki funkcje posiadają zmienne lokalne, które nie są już współdzielone z innymi wątkami. Każdy wątek posiada też swój własny stos, związany z kolejno wywoływanymi przez dany wątek funkcjami.

Ponadto, są sytuacje, w których przydatne okazuje się posiadanie przez wątki danych globalnych, które jednak byłyby prywatne dla danego wątku. Istnieją zatem specjalne mechanizmy pozwalające na tworzenie takich prywatnych danych globalnych.

Programowanie z użyciem wątków

Wątki pozwalają tworzyć programy działające współbieżnie. Współbieżność przydaje się wielokrotnie w programowaniu, na przykład:

- w różnego rodzaju serwerach obsługujących równocześnie wiele żądań,
- w programach okienkowych, gdzie aktualizacja stanu okienek, komunikacja z użytkownikiem (odczyt zdarzeń z klawiatury i myszy), obliczenia związane z funkcjami danego programu, jak również operacje I/O na urządzeniach zewnętrznych, wszystkie mogą być realizowane współbieżnie i niezależnie od siebie,
- tworzenie współbieżnych aplikacji ma sens również w związku z coraz większą liczbą procesorów, a także rdzeni i wątków, dostępnych we współczesnych komputerach; program napisany współbieżnie może być wtedy wykonywany szybciej niż program napisany jednowątkowo.

W niektórych z tych sytuacji adekwatne rozwiązanie można uzyskać również za pomocą współbieżnie wykonujących się i komunikujących się procesów, ale w pewnych sytuacjach model wątków ma zdecydowaną przewagę.

Współbieżność z użyciem wątków i procesów

Programowanie z wątkami jest podobne jak z procesami, z następującymi różnicami:

- Wszystkie wątki danego procesu wykonują ten sam program, podczas gdy procesy są kompletnie oddzielnymi obiektami, każdy z własną przestrzenią adresową, mogącymi wykonywać dowolne programy.
- Wątki mają automatycznie dostępną szybką komunikację przez zmienne globalne, która jednak wymaga synchronizacji.
- Tworzenie wątków jest tanie, a więc znacznie szybsze niż tworzenie procesów, co ma znaczenie np. w serwerach obsługujących bardzo wiele zgłoszeń na sekundę.
  - Czas potrzebny na pełne utworzenie nowego procesu (`fork()`) na współczesnych procesorach z zegarem 1.5÷2.8 GHz wynosi w granicach od 0.1 nawet do 2 ms. Czas potrzebny na utworzenie nowego wątku jest o około rząd wielkości mniejszy: 15 do 50 μs (`pthread_create()`).
  - Jednak należy pamiętać, że utworzone już wątki nie mają żadnej przewagi, i wykonują się tak samo szybko jak utworzone procesy.

Pomijając interakcje między sobą, współbieżnie wykonujące się wątki zachowują się całkiem podobnie do współbieżnie wykonujących się procesów. Są również obsługiwane podobnie przez system operacyjny w trzech stanach: wykonywany, gotowy, i oczekujący.

Wątki użytkownika

Ze względu na sposób realizacji wątków w systemie operacyjnym można wyróżnić dwa rodzaje wątków: wątki użytkownika i wątki jądra. **Wątki użytkownika** (*user level threads*, ULT) są tworzone i w całości obsługiwane w przestrzeni użytkownika, zaimplementowane samodzielnie, lub za pomocą odpowiedniej biblioteki. Wtedy:

- jądro systemu nie ma świadomości istnienia wątków, ani nie zapewnia im wsparcia,
- możliwe jest ich zastosowanie nawet w systemach nieobsługujących wątków,
- tworzenie i przełączanie wątków jest szybkie, ponieważ wykonuje tylko tyle kodu ile wymaga, bez przełączania kontekstu jądra,
- jeśli jeden z wątków wywoła funkcję systemową powodującą blokowanie (czekanie), to pozostałe wątki nie będą mogły kontynuować pracy,
- nie ma możliwości wykorzystania wielu procesorów/rdzeni do wykonywania różnych wątków programu.

Przykładem popularnej, przenośnej biblioteki wątków użytkownika dla systemów POSIX/ANSI-C jest GNU Pth (*GNU Portable Threads*).

Wątki jądra

Większość współczesnych systemów operacyjnych realizuje wątki w jądrze systemu, zwane **wątkami jądra** (*kernel level threads*, KLT). Dla programów wykorzystujących te wątki:

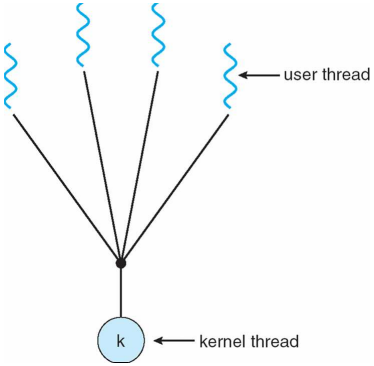
- jądro zajmuje się obsługą wątków (tworzeniem, przełączaniem); trwa to typowo dłużej niż dla wątków użytkownika,
- wątek czekający na coś w funkcji systemowej nie blokuje innych wątków,
- w systemach wieloprocesorowych jądro ma możliwość jednoczesnego wykonywania wielu wątków jednej aplikacji na różnych procesorach.

Wątki jądra wspierane są przez prawie wszystkie współczesne systemy operacyjne skali makro, i wiele systemów operacyjnych skali mikro.

Jednak ani czysty model wątków użytkownika ani czysty model wątków jądra nie zapewniają połączenia elastyczności programowania z efektywnością wykonywania programu. Dobrym modelem jest natomiast taka implementacja biblioteki wątków użytkownika, która odwzorowuje wątki użytkownika na wątki jądra.

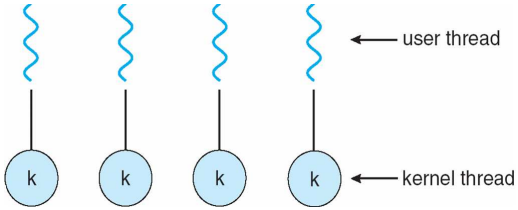
Odwzorowania wątków użytkownika na wątki jądra

Wiele-jeden



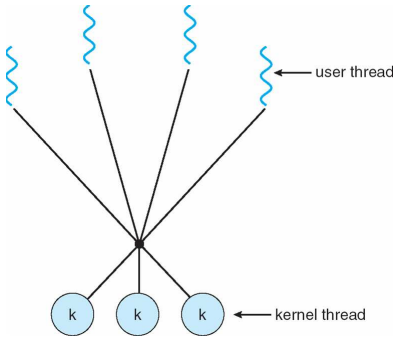
Ten model ma zastosowanie do czystej biblioteki wątków użytkownika, przenośnych bibliotekach wątków (np. GNU Portable Threads), w systemach jednoprocesorowych, lub nieobsługujących wątków.

Jeden-jeden



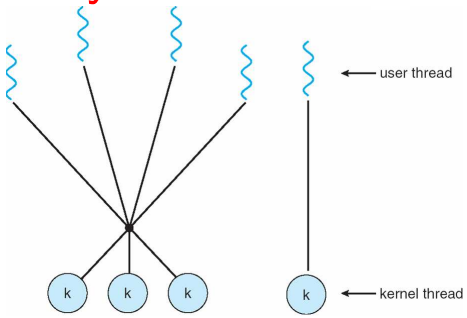
Ten model jest najczęściej obecnie stosowany, np. w systemach Solaris, Linux, Windows XP.

Wiele-wiele



Model stosowany w dawniejszych implementacjach wątków w systemach operacyjnych. Pozwala kontrolować rzeczywisty stopień współbieżności programów przez przydzielanie odpowiedniej liczby wątków jądra.

Model mieszany



Model podobny do modelu „wiele-wiele” ale pozwala na przydzielenie wątku jądra wybranym wątkom użytkownika. Stosowany np. w systemach: IRIX, HP-UX, Tru64 UNIX.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Porównaj równoległość realizowaną za pomocą procesów i wątków — wymień podobieństwa i różnice.

2. W szczególności porównaj:
   - różnice w szybkości tworzenia procesów i wątków,
   - różnice w szybkości pracy procesów i wątków,
   - różnice w szybkości komunikacji procesów i wątków przez pamięć wspólną.

3. Porównaj realizację wątków w modelu wątków jądra i użytkownika — jakie są zalety każdego modelu.

Wątki Pthread

Standard wątków POSIX (IEEE 1003.1c) zwany Pthread (lub Pthreads) jest modelem współbieżności wielowątkowej i standardem programowania aplikacji wielowątkowych. Standard wprowadza interfejs programistyczny (API), który posiada szereg implementacji dla wszystkich współczesnych systemów uniksowych, w tym Mac OS X, i Android. Istnieją również implementacje dla systemu Windows.

Standard Pthread definiuje funkcje tworzenia, zarządzania, i synchronizacji wątków. Jest zwykle zrealizowany za pomocą oddzielnej biblioteki. Kompilacja w systemach uniksowych wymaga użycia flagi `-lpthread`.

Na podstawowym poziomie reprezentuje on zatem model wątków użytkownika. Jednak funkcje Pthread pozwalają zarządzać równoległością wykonywania wątków, oraz ich szeregowaniem w systemie operacyjnym, zatem implementacja biblioteki musi odwzorować wątki Pthread na wątki jądra danego systemu. Standard Pthread realizuje zatem mieszany model implementacji wątków.

Podstawowe pojęcia modelu wątków Pthread

- Każdy proces startuje jako jednowątkowy i może w dowolnym momencie zacząć tworzyć dodatkowe wątki.
- Wątki tworzy się funkcją `pthread_create` określając funkcję (i jej argument), której wywołanie rozpoczyna wykonywanie wątku.
- Wątek główny procesu ma znaczenie nadrzędne — gdy on się kończy to kończą się wszystkie wątki, oraz cały proces.
- Istnieje zestaw atrybutów, którymi można indywidualnie kontrolować zachowanie każdego wątku. Atrybuty mają wartości domyślne, ale można je zmieniać.
- Wątek może zakończyć się sam, a także może go zakończyć inny wątek procesu funkcją `pthread_cancel`.
- Zakończenie wątku jest trochę podobne do zakończenia procesu: wątek generuje kod zakończenia (status), który może być odczytany przez wątek macierzysty.
- Jeden wątek może przejść w stan oczekiwania na zakończenie innego wątku, co zwane jest **wcielaniem** (*join*) i wykonywane funkcją `pthread_join`.
- Wątki mogą być **odłączone** i wtedy nie podlegają wcielaniu, tylko kończą się bez potrzeby odczytywania ich statusu.

Uwagi o kasowaniu wątków

- Skasowanie wątku bywa przydatne, gdy program uzna, że nie ma potrzeby wykonywać dalej wątku, pomimo iż nie zakończył on jeszcze swej pracy. Wątek kasuje funkcja `pthread_cancel()`.

- Skasowanie wątku może naruszyć spójność modyfikowanych przezeń globalnych danych, zatem powinno być dokonywane w przemyślany sposób. Istnieją mechanizmy wspomagające „oczyszczanie" struktur danych (*cleanup handlers*) po skasowaniu wątku. W związku z tym skasowany wątek nie kończy się od razu, lecz w najbliższym punkcie kasowania (*cancellation point*). Jest to tzw. kasowanie synchroniczne.

- Punktami kasowania są wywołania niektórych funkcji, a program może również stworzyć dodatkowe punkty kasowania wywołując funkcję `pthread_testcancel()`.

- W przypadkach gdy wątek narusza struktury danych przed jednym z punktów kasowania, może on zadeklarować procedurę czyszczącą, która będzie wykonana automatycznie w przypadku skasowania wątku. Tę procedurę należy oddeklarować natychmiast po przywróceniu spójności struktur danych.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jak się ma model wątków standardu Pthread do modeli wątków użytkownika i wątków jądra?

Semantyka wątków a semantyka procesów POSIX

W semantyce procesów systemu Unix, utrwalonej standardami POSIX, istnieje szereg mechanizmów, pierwotnie zaprojektowanych do modelu procesów, których semantyka koliduje z modelem wątków. Takimi mechanizmami są:

- funkcja `fork`,
- funkcje z grupy `exec*`,
- obsługa sygnałów,
- sygnalizacja błędów przez funkcje systemowe,
- ogólnie efekty globalne generowane przez niektóre funkcje systemowe.

Można zauważyć, że o ile pierwsze trzy mechanizmy są elementami konstrukcji modelu procesów POSIX, to ostatnie dwa są niefortunnymi wczesnymi koncepcjami, które nie rodziły problemów w ramach modelu procesu, jednak przeniesione do modelu procesów generują zasadnicze problemy.

Semantyka wątków: funkcja `fork`

Funkcja fork tworzy nowy proces przez klonowanie istniejącego. Ideą tej funkcji jest stworzenie współbieżności, gdy jej jeszcze nie ma. Ale co, gdy funkcję fork wywoła jeden z wątków procesu wielowątkowego? Może pojawić się wątpliwość, czy nowy proces ma mieć dokładnie te same wątki co rodzic, czy też powstać jako proces jednowątkowy?

Gdyby proces potomny miał dokładnie takie same wątki jak rodzic, to zachodzi pytanie, co robiły wątki rodzica w chwili utworzenia potomka. Niektóre mogły zapisywać plik (wtedy plik będzie w dalszym ciągu zapisywany przez dwa odpowiadające sobie wątki w dwóch procesach), niektóre mogły czekać na dane z jakiegoś interfejsu komunikacyjnego (wtedy pojawia się wątpliwość który wątek powinien otrzymać dane, gdy one się pojawią), jeszcze inne mogły wcześniej zająć jakiś semafor (wtedy po utworzeniu potomka semafor będzie jakby zajęty podwójnie, przez wątki w dwóch procesach), itp. We wszystkich tych przypadkach praca dwóch identycznych wielowątkowych procesów może rodzić nieoczekiwane problemy.

Zgodnie ze standardem POSIX, oryginalna funkcja fork tworzy proces jednowątkowy. Rozwiązaniem stosowanym w niektórych systemach jest istnienie dwóch wersji funkcji fork — jedna duplikuje wszystkie wątki, a druga tworzy proces potomny jednowątkowy. Programista może zdecydować której wersji fork użyć, i dostosować swój program do jej zachowania. System Solaris posiada w tym celu funkcje fork1 i forkall. W systemie Linux istnieje tylko wersja fork wersja tworząca proces jednowątkowy.

Semantyka wątków: funkcje `exec*`

Podobne wątpliwości można mieć w odniesieniu do funkcji grupy `exec*`. Proces w systemie Unix może wywołać jedną z funkcji `exec*` co powoduje „przeobrażenie" całego procesu i rozpoczęcie wykonywania innego programu. Zachodzi pytanie, jak takie przeobrażenie powinno wyglądać dla procesu, który utworzył już kilka wątków, i oto jeden z wątków wywołał jakąś funkcję `exec*`.

Zostało to rozwiązane w ten sposób, że w momencie wywołania jednej z funkcji `exec*` przez dowolny wątek procesu giną wszystkie jego utworzone dodatkowo wątki, proces redukuje się do jednowątkowego, i rozpoczyna wykonanie nowego programu zgodnie ze zwykłą semantyką funkcji `exec*`.

Semantyka wątków: obsługa sygnałów

- Każdy wątek ma własną maskę sygnałów. Wątek dziedziczy ją od wątku, który go zainicjował, lecz nie dziedziczy sygnałów czekających na odebranie.
- Sygnał wysłany do procesu jest doręczany do jednego z jego wątków.
- Sygnały synchroniczne, tzn. takie, które powstają w wyniku akcji danego wątku, np. SIGFPE, SIGSEGV, są doręczane do wątku, który je wywołał.
- Sygnał do określonego wątku można również skierować funkcją `pthread_kill`.
- Sygnały asynchroniczne, które powstają bez związku z akcjami, np. SIGHUP, SIGINT, są doręczane do jednego z tych wątków procesu, które nie mają tego sygnału zamaskowanego.
- Jedną ze stosowanych konfiguracji obsługi sygnałów w programach wielowątkowych jest oddelegowanie jednego z wątków do obsługi sygnałów, i maskowanie ich we wszystkich innych wątkach.

Semantyka wątków: sygnalizacja błędów

Model sygnalizacji błędów przez funkcje systemowe Uniksa polega na:

1. sygnalizacji wystąpienia błędu w funkcji systemowej przez zwrócenie pewnej specyficznej wartości, która może być różna dla różnych funkcji (typowo: dla funkcji zwracających `int` jest to wartość -1, a dla funkcji zwracających wskaźnik jest to wartość `NULL`),
2. ustawienia wartości zmiennej globalnej procesu `errno` na kod zaistniałego błędu; należy pamiętać, że wartość zmiennej `errno` pozostaje ustawiona nawet gdy kolejne wywołania funkcji systemowych są poprawne.

W oczywisty sposób, model ten nie będzie działał poprawnie w programach z wątkami. Jednocześnie wprowadzenie zupełnie innego modelu w systemach uniksowych wymagałoby przedefiniowania wszystkich funkcji, i jest nie do pomyślenia. Wprowadzono więc następujące zasady:

- funkcje związane z wątkami nie sygnalizują błędów przez zmienną `errno` (która jest globalna w procesie), lecz przez niezerowe wartości funkcji,
- dla umożliwienia „zwykłym" funkcjom sygnalizacji przyczyn błędów, każdy z wątków MOŻE otrzymać prywatną kopię zmiennej globalnej `errno` (wymaga to kompilacji programu z makrodefinicją `-D_REENTRANT`).

Semantyka wątków: efekty globalne w funkcjach systemowych

Poza wyżej omówionymi, szereg dalszych funkcji systemu Unix generuje globalne efekty uboczne, np. w celu korzystania z nich w dalszych wywołaniach. Funkcje te zostały zdefiniowane dawniej, zanim powstały wątki, i są one po prostu niekompatybilne z użyciem wątków w programach (np. funkcja `strtok` biblioteki `string`).

Funkcje, które nie tworzą takich efektów, i mogą być bez ograniczeń używane w programach z wątkami, nazywane są: *thread-safe* lub *reentrant*. Ta druga nazwa określa fakt, że funkcja może być jednocześnie wiele razy wywoływana w jednej przestrzeni adresowej, i odnosi się to zarówno do wywołań w wątkach, jak i wywołań rekurencyjnych.

Dla funkcji, które nie są *thread-safe* albo *reentrant*, zdefiniowano w systemach uniksowych ich zamienniki, które mają te własności. Zamienniki mają inne nazwy (np. `strtok_r`) i inny interfejs wywołania. W programach z wątkami należy uważać na te kwestie, i wykorzystywać tylko funkcje *thread-safe*.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jakie elementy modelu procesów Unix/POSIX są niekompatybilne z semantyką wątków?