Wątki

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

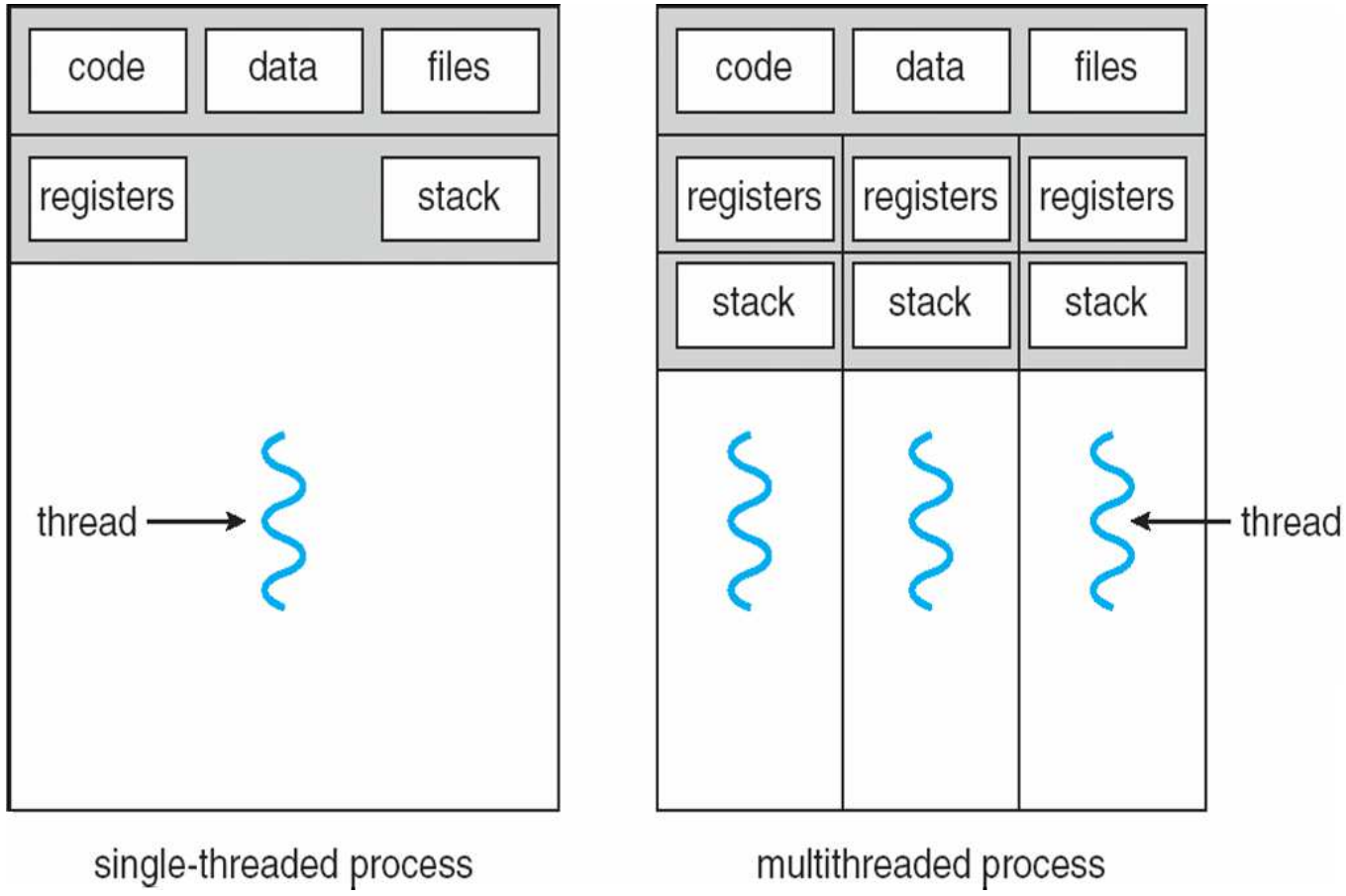
- kod programu załadowany do pamięci operacyjnej, wraz ze środowiskiem (zbiorem zmiennych posiadających wartości) i zasobami (np. zbiorem otwartych plików, i innymi przydzielonymi zasobami, np. portami komunikacyjnymi, itd.) przechowywanym przez jądro systemu,

- 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.

Przydatne okazują się również jednak mechanizmy pozwalające na tworzenie danych globalnych, które byłyby prywatne dla danego wątku.

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, np. oczekująca na odczyt danych z urządzenia, może być realizowana asynchronicznie do pętli oczekującej na reakcję użytkownika; tego rodzaju oczekiwanie na kilka różnych rzeczy naraz stwarza trudność w programowaniu,
- tworzenie współbieżnych aplikacji ma sens również w związku z coraz większą liczbą procesorów (lub rdzeni) dostępnych we współczesnych komputerach; program napisany współbieżnie może być wtedy wykonywany szybciej niż program napisany jednowątkowo.

Współbieżność z użyciem wątków i procesów

Programowanie z wątkami jest podobne do programowania procesów, 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 (w sensie szybkości), co ma znaczenie np. w serwerach obsługujących bardzo wiele zgłoszeń na sekundę.

Pomijając interakcje, współbieżnie wykonujące się wątki zachowują się całkiem podobnie do współbieżnie wykonujących się procesów.

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.

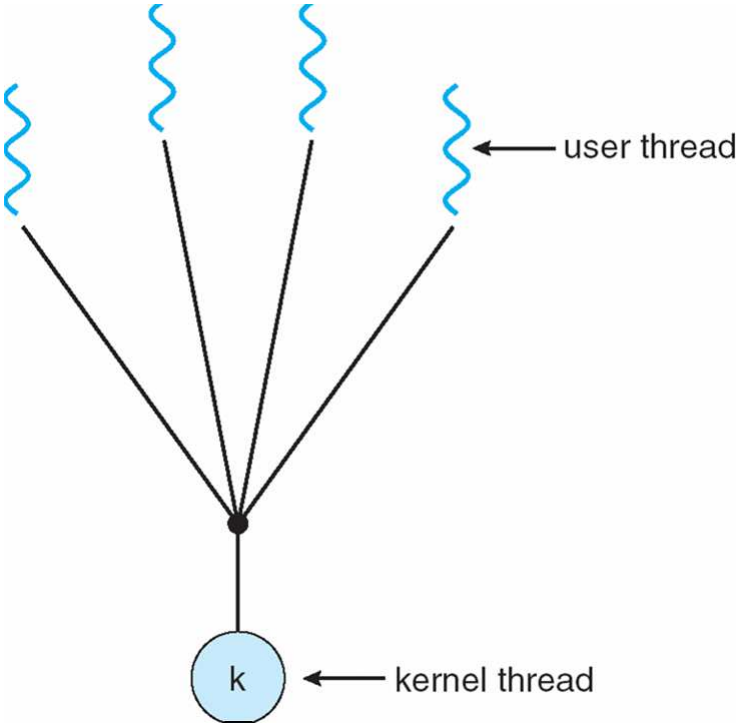
Wątki jądra

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.

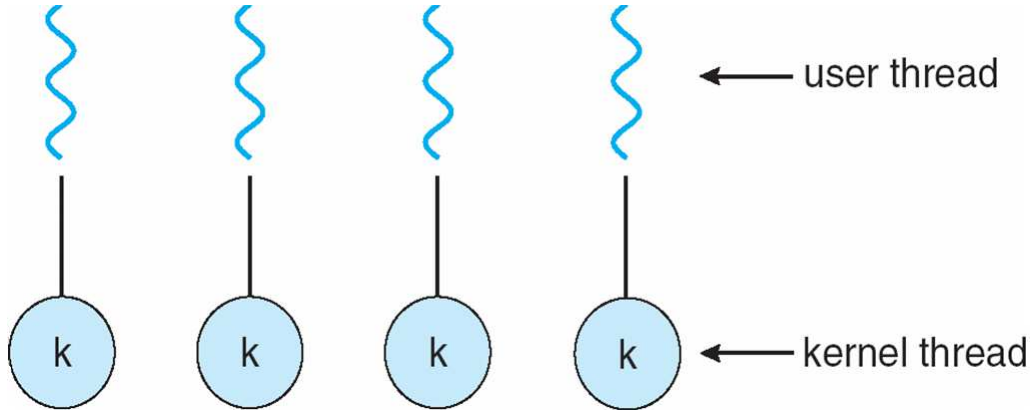
Ani czysty model wątków użytkownika ani czysty model wątków jądra nie zapewniają połączenia elastyczności programowania z efektywnością wykonywania programu. Dobrym modelem jest natomiast taka implementacja biblioteki wątków użytkownika, która odwzorowuje wątki użytkownika na wątki jądra.

Odwzorowanie wiele-jeden



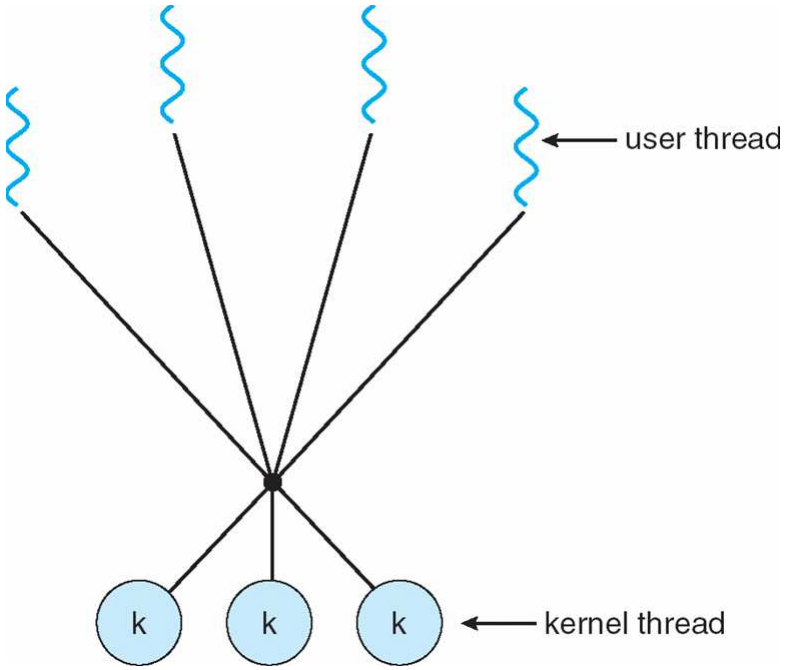
Ten model ma zastosowanie do czystej biblioteki wątków użytkownika, przenośnych bibliotekach wątków (np. GNU Portable Threads), w systemach jednoprocesorowych, lub nieobsługujących wątków.

Odwzorowanie jeden-jeden



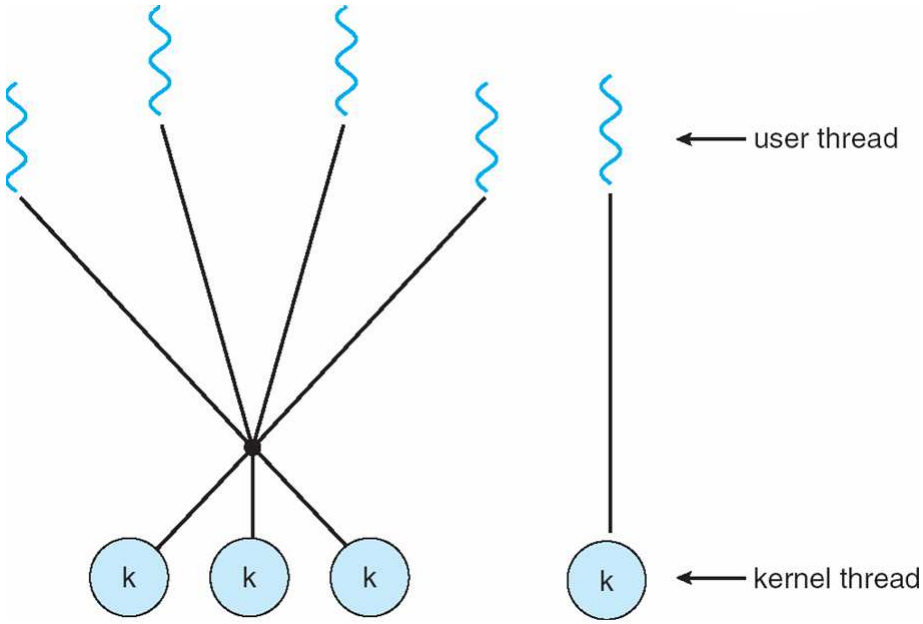
Ten model jest najczęściej obecnie stosowany, np. w systemach Solaris, Linux, Windows XP.

Odwzorowanie 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 POSIX wątków Pthread (IEEE 1003.1c) definiuje interfejs tworzenia i zarządzania wątkami dla programów. Standard ten posiada szereg implementacji dla wszystkich współczesnych systemów uniksowych. Istnieją również implementacje dla systemu Windows. Standard Pthread jest zwykle zrealizowany za pomocą odpowiedniej biblioteki. Najważniejsze pojęcia:

- wątki tworzy się funkcją `pthread_create` określając funkcję (i jej argument), której wywołanie rozpoczyna wykonywanie wątku,
- wątki posiadają zestaw atrybutów; które 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,
- wątek główny procesu ma znaczenie nadrzędne — gdy on się kończy to kończą się wszystkie wątki procesu.

Wątki Pthread a semantyka procesów

W systemie Unix istnieje rozbudowana lecz częściowo niefortunna semantyka procesów, której szereg elementów koliduje z semantyką wątków. Takimi elementami są: funkcje `exec*`, sygnalizacja błędów przez funkcje systemowe (i ogólnie efekty globalne generowane przez niektóre funkcje systemowe), oraz sygnały.

Wątki Pthread: funkcje `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, i proces przeobraża się tak, jak tradycyjny proces jednowątkowy, rozpoczynając wykonanie nowego programu zgodnie ze zwykłą semantyką funkcji `exec*`.

Wątki Pthread: 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`).

Wątki Pthread: efekty globalne w innych funkcjach

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*.

Wątki Pthread: 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.

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

Synchronizacja wątków

- Ponieważ wątki są z definicji wykonywane współbieżnie i asynchronicznie, a także operują na współdzielonych globalnych strukturach danych, konieczne są mechanizmy dla zapewnienia wyłączności dostępu.

- Mechanizmy wzajemnej synchronizacji i blokowania ograniczają współbieżność, zatem należy stosować jak najdrobniejsze blokady (patrz dalej).

- Dostępne mechanizmy synchronizacji standardu POSIX
  - mutexy (*mutual exclusion*)
  - blokady odczytu i zapisu (*read-write locks*)
  - semafory
  - zmienne warunkowe (*conditional variables*)
  - bariery

Synchronizacja wątków: muteksy

Muteksy służą do wprowadzenia ochrony fragmentu programu, zwanego **sekcją krytyczną**, która może wprowadzić błędy jeśli będzie wykonana równocześnie przez różne procesy/wątki. Rozważmy wielowątkowy program obsługujący operacje na dowolnych kontach bankowych, spływające z oddziałów i bankomatów:

Wątek 1:
```
// otrzymany numer konta
// oraz wartosc operacji
saldo[konto] += operacja;
```

Wątek 2:
```
// otrzymany numer konta
// oraz wartosc operacji
saldo[konto] += operacja;
```

Tak równoległe jak i quasi równoległe wykonanie tych wątków może spowodować takie pofragmentowanie tych operacji, że jeśli jednocześnie pojawią się dwie operacje na tym samym koncie, to zostanie obliczona niepoprawna wartość salda. Zjawisko takie jest nazywane **wyścigami**. Można mu zapobiec za pomocą muteksu:

Wątek 1:
```
pthread_mutex_lock(transakcyjny);
saldo[konto] += operacja;
pthread_mutex_unlock(transakcyjny);
```

Wątek 2:
```
pthread_mutex_lock(transakcyjny);
saldo[konto] += operacja;
pthread_mutex_unlock(transakcyjny);
```

Założenie blokady muteksu powoduje, że żadna z tych operacji nie może być przerwana przez drugą. W danej chwili tylko jeden wątek może posiadać blokadę muteksu, a drugi będzie musiał na nią czekać.

Synchronizacja wątków: muteksy (cd.)

Ważną kwestią związaną z operowaniem na zmiennych synchronizacyjnych przez wielowątkowe programy jest, że operacje te muszą być realizowane przez system operacyjny, a nie program użytkownika. Program, który sprawdziłby, że wartość muteksu wynosi 0 (wolny), i chciałby następnie ustawić go na 1 (zajęty), mógłby zostać wywłaszczony zaraz po sprawdzeniu, i w wyniku wyścigów zająć muteks już w międzyczasie zajęty przez inny wątek.

Jedynie system operacyjny może zapewnić nierozdzielne wykonanie sprawdzenia i zajęcia muteksu. Każdy mechanizm synchronizacji ma zatem swój interfejs funkcyjny udostępniający właściwy zestaw operacji. Na przykład, podstawowe operacje na muteksach (z pominięciem operacji na atrybutach muteksów):

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Synchronizacja wątków: muteksy (cd.)

Muteks, podobnie jak inne mechanizmy synchronizacji zdefiniowane przez standard POSIX, wykorzystują struktury pamięciowe tworzone jawnie w programach. Muteks jest zwykłą zmienną logiczną, ustawianą na 1 (blokada) lub 0 (brak blokady), plus informacja, który wątek założył blokadę muteksu, i zatem ma prawo ją zdjąć.

Przy deklaracji zmiennych typu muteks należy pamiętać, żeby zainicjalizować je na zero, co oznacza muteks odblokowany.

```
static pthread_mutex_t mutex1;

pthread_mutex_t *mutex2;
mutex2 = (pthread_mutex_t *)
         calloc(1, sizeof(pthread_mutex_t));

pthread_mutex_t mutex3 = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t mutex4;
  pthread_mutexattr_t attr_ob;
  pthread_mutexattr_init(&attr_ob)
  pthread_mutex_init(&mutex4, &attr_ob)
```

Synchronizacja wątków: blokady zapisu i odczytu

Blokady zapisu i odczytu są innym rodzajem mechanizmu synchronizacyjnego, będącym pewnym uogólnieniem muteksu. Wyobraźmy sobie, że wśród napływających operacji bankowych większość stanowi sprawdzenie salda. Pobranie salda nie może być wykonywane równocześnie z jakąkolwiek operacją wpłaty ani wypłaty, ponieważ w tym czasie saldo mogłoby być niepoprawnie odczytane.

Jednak nic nie stoi na przeszkodzie, żeby sprawdzanie salda nie mogło się odbywać równocześnie z innymi operacjami sprawdzenia salda, nawet tego samego konta. Aby to zrealizować, do zabezpieczenia obsługi transakcji zamiast muteksu należy użyć blokady zapisu i odczytu. Przy realizacji transakcji wpłaty lub wypłaty zakładana będzie blokada zapisu, co wyklucza wszelkie inne jednoczesne operacje. Jednak przy realizacji jakiejkolwiek operacji sprawdzenia salda, należy użyć blokady odczytu, co powoduje, że inne operacje sprawdzenia salda mogą przebiegać równolegle, a wstrzymywane będą tylko operacje wpłaty i wypłaty.

Korzyść z użycia tego mechanizmu będzie znacząca tylko wtedy gdy operacji wymagających blokad odczytu będzie znacznie więcej niż tych wymagających blokad zapisu. W wielu systemach tak jest, np. w systemie weryfikacji haseł lub PINów będzie znacznie więcej operacji odczytu (sprawdzenia) niż operacji zapisu (zmiany) hasła/PINu.

Operacje na blokadach zapisu i odczytu:

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Synchronizacja wątków: drobnoziarnistość

Zauważmy, że mechanizmy synchronizacji, jakkolwiek potrzebne, ograniczają współbieżność działania wielowątkowych programów. Wiele wątków może zostać wstrzymanych na muteksie lub blokadzie założonej przez pojedynczy wątek.

Istnienie blokad zapisu i odczytu jest wyrazem ogólnej tendencji do tworzenia **drobnoziarnistych** mechanizmów synchronizacji. Program realizujący zmianę PINu użytkownika mógłby założyć blokadę bazy PINów na cały swój czas działania, na czas wykonania jednej procedury zawierającej zapis PINu, lub tylko dokładnie na czas zapisywania nowegu PINu. Ta ostatnia możliwość stanowi synchronizację najbardziej drobnoziarnistą i w najmniejszym stopniu ogranicza współbieżność.

Blokady zapisu i odczytu są uogólnieniem muteksów, ale logicznie dają to samo: prawidłową synchronizację pracy wielu wątków. Zauważmy, że założenie blokady zapisu jest praktycznie równoważne założeniu blokady zwykłego muteksu. Fakt, że istnieje oddzielny mechanizmu blokady odczytu pozwala zwiększyć współbieżność programu, dzięki drobnoziarnistości synchronizacji.

Synchronizacja wątków: semafory

Semafory są innym uogólnieniem muteksów, pozwalającym zarządzać zasobami, mierzonymi w sztukach.[1] Zamiast w całości zajmować dany zasób, wątki zajmują pewną liczbę jednostek tego zasobu, w granicach aktualnie dostępnej puli. Wątek, który zażąda zajęcia większej liczby niż aktualnie dostępna musi czekać, aż inny wątek/inne wątki nie zwolnią dostatecznej liczby jednostek.

Analogią, ilustrującą zastosowanie semaforów może być wystawa (np. malarstwa), na którą ze względów bezpieczeństwa można wpuścić jednorazowo maksymalnie N osób. Semafor wstępnie ustawiamy na wartość N. Każda wchodząca osoba zgłasza żądanie jednostkowego zajęcia semafora, co dekrementuje jego licznik (wyrażający liczbę wolnych miejsc na wystawie). Po osiągnięciu maksimum (wyzerowaniu semafora), następne osoby zgłaszające żądania zajęcia semafora muszą czekać. Każda wychodząca osoba zwalnia (inkrementuje) semafor, co może spowodować wpuszczenie jednej osoby do środka, o ile czeka w żądaniu zajęcia semafora.

[1] Jak to było już wyjaśnione wcześniej, nazwa semafor jest myląca. W kolejnictwie semafor jest stosowany do sygnalizacji zajętego toru. Informatycznym odpowiednikiem tego mechanizmu jest mutex. Semafory wprowadził w 1965 holenderski informatyk Edsger Dijkstra. Pojęcie muteksu, jako semafora binarnego, zostało zdefiniowane dużo później.

Muteks można traktować jako szczególny przypadek semafora. Logicznie muteks jest równoważny semaforowi przyjmującemu tylko wartości 1 lub 0. Zatem semafory są uogólnienie muteksów, ponieważ dostarczają większych możliwości.

Jednak jest to uogólnienie innego rodzaju niż blokady zapisu i odczytu, które logicznie dostarczają tych samych usług co muteksy. Wprowadzając rozdzielenie typu blokady, pozwalają one na zwiększenie drobnoziarnistości synchronizacji.

Synchronizacja wątków: semafory — implementacja

Biblioteka Pthread nie wprowadza semaforów, ponieważ istnieje szereg innych standardów definiujących semafory, dobrze implementowanych we wszystkich systemach. Poniżej przedstawiono interfejs funkcyjny semaforów standardu POSIX Realtime. Mechanizmy tego standardu różnią się od mechanizmów Pthread między innymi tym, że są oparte na systemie plików. Semafory tworzone w tym standardzie istnieją jako pliki w systemie plików, aczkolwiek standard nie określa jaki ma to być system plików. Na przykład, może to być wirtualny system plików, niezlokalizowany na dysku komputera.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *restrict sem, int *restrict sval);

int sem_init(sem_t *sem, int pshared, unsigned int value);
sem_t *sem_open(const char *name, int oflag,
                /* unsigned long mode, unsigned int value */ ...);
int sem_close(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Synchronizacja wątków: zmienne warunkowe

Zmienne warunkowe służą do czekania na, i sygnalizowania, spełnienia jakichś warunków. Ogólnie schemat użycia zmiennej warunkowej ze stowarzyszonym muteksem jest następujący:

```
struct {
  pthread_mutex_t mutex;
  pthread_cond_t  cond;
  { ... inne zmienne przechowujace warunek }
} global = { PTHREAD_MUTEX_INITIALIZER,
             PTHREAD_COND_INITIALIZER, {...} };
```

schemat sygnalizacji zmiennej warunkowej:

```
pthread_mutex_lock(&global.mutex);
// ustawienie oczekiwanego warunku
pthread_cond_signal(&global.cond);
pthread_mutex_unlock(&global.mutex);
```

schemat sprawdzania warunku i oczekiwania:

```
pthread_mutex_lock(&global.mutex);
while ( { warunek niespelniony } )
  pthread_cond_wait(&global.cond,
                    &global.mutex);
//wykorzystanie oczekiwanego warunku
pthread_mutex_unlock(&global.mutex);
```

Synchronizacja wątków: zmienne warunkowe (cd.)

Pełny interfejs funkcjonalny zmiennych warunkowych:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Algorytm działania funkcji `pthread_cond_wait`:

1. odblokowanie mutexu podanego jako argument wywołania
2. uśpienie wątku do czasu aż jakiś inny wątek wywoła funkcję `pthread_cond_signal` na danej zmiennej warunkowej
3. ponowne zablokowanie mutexu

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Dlaczego potrzebne są mechanizmy synchronizacji?

2. Czy dodanie synchronizacji do programu wielowątkowego może raczej przyspieszyć czy spowolnić jego wykonanie?

3. Wyjaśnij mechanizm ochrony sekcji krytycznej programu muteksem.

4. Wyjaśnij dlaczego operacje na muteksie, który jest zwykłą zmienną logiczną, muszą być realizowane przez system operacyjny, a nie przez program użytkownika.

5. Wyjaśnij dlaczego drobnoziarnistość synchronizacji zwiększa współbieżność programów.

6. Porównaj ochronę sekcji krytycznej muteksem i blokadą zapisu i odczytu.

7. Porównaj operacje dostępne na muteksach i semaforach.

8. Opisz współpracę wątków synchronizowaną za pomocą zmiennej warunkowej.

Przykład: producenci i konsumenci

Przykładowy program producentów i konsumenta (jednego), zaczerpnięty z książki Stevensa (nowe wydanie, tom 2). Pierwsza wersja nie zawiera żadnej synchronizacji operacji na zmiennych globalnych i może generować błędy.

```
#define MAXNITEMS       1000000
#define MAXNTHREADS         100

int     nitems; /* read-only by producer and consumer */
struct {
  pthread_mutex_t   mutex;
  int   buff[MAXNITEMS];
  int   nput;
  int   nval;
} shared = { PTHREAD_MUTEX_INITIALIZER };

void    *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
  int         i, nthreads, count[MAXNTHREADS];
  pthread_t   tid_produce[MAXNTHREADS], tid_consume;
```

```
  if (argc != 3)
    err_quit("usage: prodcons2 <#items> <#threads>");
  nitems = min(atoi(argv[1]), MAXNITEMS);
  nthreads = min(atoi(argv[2]), MAXNTHREADS);

  Set_concurrency(nthreads);
  /* start all the producer threads */
  for (i = 0; i < nthreads; i++) {
    count[i] = 0;
    Pthread_create(&tid_produce[i], NULL, produce,
                   &count[i]);
  }

  /* wait for all the producer threads */
  for (i = 0; i < nthreads; i++) {
    Pthread_join(tid_produce[i], NULL);
    printf("count[%d] = %d\n", i, count[i]);
  }

  /* start, then wait for the consumer thread */
  Pthread_create(&tid_consume, NULL, consume, NULL);
  Pthread_join(tid_consume, NULL);

  exit(0);
}
```

```
void *
produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (shared.nput >= nitems) {
      Pthread_mutex_unlock(&shared.mutex);
      return(NULL); /* array is full, we're done */
    }
    shared.buff[shared.nput] = shared.nval;
    shared.nput++;
    shared.nval++;
    Pthread_mutex_unlock(&shared.mutex);
    *((int *) arg) += 1;
  }
}

void *
consume(void *arg) {
  int     i;
  for (i = 0; i < nitems; i++) {
    if (shared.buff[i] != i)
      printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```

Przykład: producenci i konsumenci (2)

Druga wersja programu z konsumentem pracującym równolegle i synchronizacją wszystkich operacji na zmiennych globalnych za pomocą muteksu. Wykorzystanie muteksu powoduje całkowicie poprawne działanie programu.

Fragment funkcji `main` uruchamiający wszystkie wątki równolegle:

```
Set_concurrency(nthreads + 1);
/* create all producers and one consumer */
for (i = 0; i < nthreads; i++) {
  count[i] = 0;
  Pthread_create(&tid_produce[i], NULL, produce,
                 &count[i]);
}
Pthread_create(&tid_consume, NULL, consume, NULL);
```

```
void *
produce(void *arg)
{
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (shared.nput >= nitems) {
      Pthread_mutex_unlock(&shared.mutex);
      return(NULL); /* array is full, we're done */
    }
    shared.buff[shared.nput] = shared.nval;
    shared.nput++;
    shared.nval++;
    Pthread_mutex_unlock(&shared.mutex);
    *((int *) arg) += 1;
  }
}
```

```
void
consume_wait(int i)
{
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (i < shared.nput) {
      Pthread_mutex_unlock(&shared.mutex);
      return;          /* an item is ready */
    }
    Pthread_mutex_unlock(&shared.mutex);
  }
}

void *
consume(void *arg)
{
  int     i;

  for (i = 0; i < nitems; i++) {
    consume_wait(i);
    if (shared.buff[i] != i)
      printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```