# POSIX Threads

Wojciech Muła

marzec 2010 z późniejszymi poprawkami
(plik utworzony 21 marca 2011)

**Najbardziej aktualna** wersja online jest utrzymywana w serwisie Wikibooks.
Wersja PDF jest dostępna adresem 0x80.pl.

Skład wykonano programem LaTeX 2ε

# Rozdział 1

# Wiadomości wstępne

Niniejszy podręcznik ma na celu zaznajomienie Czytelnika z biblioteką programistyczną **POSIX Threads** (w skrócie **pthreads**). Biblioteka udostępnia jednolite API dla język C do tworzenia i zarządzania wątkami , jest dostępna w systemach m.in. Linux , FreeBSD , Windows .

## 1.1 Spis treści

## 1.2 Podstawowe informacje o bibliotece

Specyfikacja POSIX jest dostępna nieodpłatnie na stronie organizacji **Open Group** (http://www.opengroup.org/). Dla wygody Czytelnika linkujemy w tym podręczniku do konkretnych podstron opisujących funkcje.

Opis części implementacji dla systemu Linux znajduje się na stronie **The Linux man-pages project** (http://www.kernel.org/doc/man-pages/).

> Niektóre strony podręczników zawierają przykładowe programy.

### 1.2.1 Pliki nagłówkowe

Podstawowy plik nagłówkowy biblioteki POSIX Threads nazywa się `pthread.h` i jego włączenie do programu jest konieczne. Dokładny wykaz funkcji znajdujący się w tym pliku, wraz z adnotacjami opcji, znajduje się na stronie http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html.

Oprócz niego:

- `sys/types.h` — typy danych (domyślnie włączane przez `pthreads.h`)
- `limits.h` — stałe określające ograniczenia biblioteki

### 1.2.2 Konwencje nazewnicze

Nazwy funkcji i typów zaczynają się zawsze od `pthread_`, stałe są pisane wielkimi literami i podobnie rozpoczynają od `PTHREAD_`. Nazewnictwo poza tym jest bardzo konsekwentne, podobnie kolejność argumentów itd.

### 1.2.3 Typy danych

Typy definiowane przez bibliotekę powinny być traktowane jako abstrakcyjne — jedynym sposobem ich inicjalizacji, zmiany wartości, itd. jest użycie **dedykowanych funkcji**.

> **UWAGA**
>
> Nie można po zainicjowaniu obiektu utworzyć jego kopii np. wykonując funkcję `memcpy`, ponieważ biblioteka może równolegle utrzymywać jakieś dodatkowe informacje dotyczące danego obiektu.

Nazwy funkcji tworzących obiekty są zgodne ze schematem `pthread_XXX_init`, natomiast niszczące obiekt `pthread_XXX_destroy`.

### 1.2.4 Atrybuty

Wszelkie dodatkowe parametry obiektów tworzonych w bibliotece są opisywane **atrybutami**, tj. innymi obiektami, które przechowują określone parametry. Obiekty atrybutów mogą być używane wielokrotnie do tworzenia różnych obiektów.

### 1.2.5 Zgłaszanie błędów

Większość funkcji z **pthreads** zwraca wartości typu `int`, która określa status operacji. Jeśli jest równy zero, funkcja wykonała się poprawnie, w przeciwnym razie zwracana jest standardowa wartość błędu (w rodzaju `EINVAL`, `ENOMEM`, `EBUSY`).

Można więc używać funkcji `strerror`, ewentualnie przypisywać wynik do zmiennej `errno` i korzystać z funkcji `perror`. Standard POSIX określa, że `errno` w środowisku wielowątkowym jest lokalne względem wątku.

Na przykład:

```
\begin{enumerate}

\item include <string.h>
\item include <errno.h>
\end{enumerate}
int main() {
int status;

status = pthread_XXX(...);
if (status)
printf("Błąd przy wywoływaniu pthread_XXX: %s", strerror(status));

errno = pthread_XXX(...);
if (errno)
perror("pthread_XXX");

return 0;
}
```

### 1.2.6 Dane użytkownika

Wszystkie dane użytkownika są przekazywane przez wskaźniki typu `void*`. O interpretacji wskaźników i danych na które wskazują decyduje wyłącznie programista, biblioteka **pthreads** nie określa ani nie ogranicza tego w żaden sposób.

### 1.2.7 Kompilacja

Ponieważ **pthreads** jest zewnętrzną biblioteką należy linkerowi podać ścieżkę do pliku bibliotecznego. Np. przy kompilacji `gcc` należy dodać opcję `-lpthread`.

## 1.3 Opcje standardu

Implementacja biblioteki **pthreads** nie musi dostarczać wszystkich funkcji opisanych w standardzie, dopuszcza on szereg opcjonalnych rozszerzeń. W tym podręczniku posługujemy się skrótami używanymi na stronach **Open Group**:

- rozszerzenia (**XSI**)
  - zmiana/odczyt rozmiaru zabezpieczającego stosu
  - wybór typu mutexu
  - zmiana/odczyt stopnia współbieżności
- bariery (**BAR**)
- wirujące blokady (**SPI**)
- zmiana priorytetu wątku posiadającego blokadę (**TPP** — Thread Priority Protection, **TPI** — Thread Priority Inheritance)
- szeregowanie wątków (**TPS** — Thread Execution Scheduling)

- synchronizacja między wątkami różnych procesów (**TSH** — Thread Process-Shared Synchronization)
- dodatkowy algorytm szeregowania dla zadań aperiodycznych w systemach czasu rzeczywistego (**TPS** — Thread Sporadic Server)
- Rozmiar i adres stosu (**TSS** — Thread Stack Size Attribute)
- Rozmiar i adres stosu (**TSA** — Thread Stack Address Attribute)
- możliwość wyboru zegara odmierzającego czas przy oczekiwaniu na zmienną warunkową; domyślnie używany jest zegar systemowy (**CS** — Clock Selection)
- czas procesora zużyty przez wątek (**TCT** — Thread CPU-Time Clocks)
- możliwość ograniczonego czasowo oczekiwania na uzyskanie blokad (mutexy) oraz blokad do odczytu/zapisu (**TMO** — Timeouts)

## 1.4 C++

Użycie biblioteki **pthreads** w programach pisanych w języku C++ jest oczywiście możliwe. Należy jednie w procedurach wykonywanych w wątkach (patrz Tworzenie wątku) obsługiwać **wszystkie** wyjątki:

```
void* wątek(void* arg) {
try {
// ...
// treść wątku
// ...
}
catch (wyjątek1) {
// obsługa wyjątku 1
}
catch (wyjątek2) {
// obsługa wyjątku 2
}
catch (...) {
// wszystkie pozostałe wyjątki
}
}
```

**Uwaga!** Przerywanie wątków w implementacji NPTL jest realizowane poprzez zgłoszenie wyjątku — jeśli w wątku zostanie użyte `catch (...)`, wówczas program zakończy się komunikatem `FATAL: exception not rethrown`. Urlich Drepper [http://udrepper.livejournal.com/ wyjaśnia na swoim blogu], jak objeść ten problem.

## 1.5 O podręczniku

### 1.5.1 Przykłady

Przykładowe programy mają jedynie na celu **zilustrowanie** pewnych cech biblioteki. Jednocześnie są to w pełni funkcjonalne programy — każdy może je skopiować do swojego komputera, skompilować i uruchomić. Mają być ułatwieniem dla własnych eksperymentów lub testów.

### 1.5.2 Autorzy

- Wojciech Muła

# Rozdział 2

# Podstawy

## 2.1 Tworzenie wątku

W programie korzystającym z biblioteki **pthreads**, tuż po uruchomieniu działa dokładnie jeden wątek, wykonujący funkcję `main`.

Aby utworzyć nowe wątki programista podaje funkcję (dokładniej: adres funkcji), która ma zostać w nim wykonana — `pthread_create` tworzy i **od razu** uruchamia wątek, oraz zwraca jego identyfikator. Oczywiście jedna funkcja może być wykorzystywana do utworzenia wielu wątków. Funkcja użytkownika przyjmuje i zwraca wartość typu `void*`, interpretacja tych danych jest zależna od programu, **pthreads** nie narzuca więc tutaj żadnych ograniczeń. Wątki mogą być tworzone z poziomu dowolnego innego wątku.

Wątek identyfikuje wartość `pthread_t`, która jest argumentem dla większości funkcji bibliotecznych.

Liczba wątków, jakie można utworzyć jest ograniczona przez stałą `PTHREAD_THREADS_MAX` (z pliku `limits.h`) oraz rzecz jasna przez zasoby systemowe.

Infobox—W Cygwinie `PTHREAD_THREADS_MAX` nie jest zdefiniowane, ponieważ system MS Windows pozwala utworzyć dowolną liczbę wątków

### 2.1.1 Typy

- `pthread_t`
  - identyfikator wątku
- `pthread_attr_t`
  - atrybutów wątku

### 2.1.2 Funkcje

- `int pthread_create(pthread_t *id, const pthread_attr_t *attr, void* (fun*)(void*), void* arg)`
  - `id` — identyfikator wątku;
  - `attr` — wskaźnik na atrybuty wątku, określające szczegóły dotyczące wątku; można podać `NULL`, wówczas zostaną użyte domyślne wartości;
  - `fun` — funkcja wykonywana w wątku; przyjmuje argument typu `void*` i zwraca wartość tego samego typu;
  - `arg` — przekazywany do funkcji.

### 2.1.3 Przykład

Poniższy program typu *hello world* tworzy (w wątku głównym) kilka innych wątków, wykonujących funkcję `watek` i czeka na ich zakończenie. Używane są domyślne atrybuty wątku.

⇒ *Przejdź do przykładowego programu nr 1.*

## 2.2 Identyfikator wątku

Wewnętrzna struktura typu `pthrad_t` (podobnie jak innych używanych w **pthreads**) jest określona przez implementację. Jedyną operacją, jaką można wykonać na identyfikatorach jest ich porównanie ze względu na równość funkcją `pthread_equal`.

Funkcja `pthread_self` zwraca identyfikator wywołującego wątku, co czasem jest przydatne.

### 2.2.1 Funkcje

- `int pthread_equal(pthread_t id1, pthread_t id2)`
  - stwierdzenie, czy identyfikatory wątków są równe
- `pthread_t pthread_self()`
  - zwraca identyfikator wywołującego wątku

## 2.3 Kończenie wątku

Wątek jest kończony w chwili, gdy funkcja użytkownika przekazana w `pthread_create` zwraca sterowanie do wywołującego ją kodu, a więc w miejscu wystąpienia instrukcji `return`.

Ponadto istnieje funkcja `pthread_exit`, która powoduje zakończenie wątku — może zostać użyta w funkcjach wywoływanych z funkcji wątku.

### 2.3.1 Przykład

W przykładowym programie zakończenie wątku powoduje wywołanie `pthread_exit` na 6. poziomie zagnieżdżenia funkcji `koniec_watku`.

⇒ *Przejdź do przykładowego programu nr 3.*

Wyjście:

```
$ ./przyklad
licznik = 0, limit = 5
 licznik = 1, limit = 5
  licznik = 2, limit = 5
   licznik = 3, limit = 5
    licznik = 4, limit = 5
     licznik = 5, limit = 5
```

## 2.4 Oczekiwanie na zakończenie wątku

Oczekiwanie na zakończenie wątku jest dodatkowym sposobem synchronizacji między wątkami i dotyczy **wyłącznie** wątków typu *joinable* (zobacz Rodzaje wątków poniżej). Wątek wywołujący `pthread_join` zostaje wstrzymany do chwili, gdy wskazany wątek zakończy działanie — wówczas wątek oczekujący jest kontynuowany, a `pthread_join` przekazuje wartość wynikową zwróconą przez wątek. Jeśli wątek został przerwany wartością zwracaną jest stała `PTHREAD_CANCELED`.

> **UWAGA**
>
> Jeśli wątek "zapętli się", oczekujący na jego zakończenie inny wątek nie jest już w stanie nic zrobić.

Oczekiwanie na zakończenie wątku można uzyskać zmiennymi warunkowymi (dodatkowo bez ograniczań na rodzaj wątku), wymaga to jednak dodatkowej pracy ze strony programisty.

### 2.4.1 Funkcje

- `int pthread_join(pthrea_t id, void **retval)`
  - `id` — identyfikator wątku, którego zakończenie jest oczekiwane;
  - `retval` — wskaźnik na wartość wynikową wątku; może być `NULL`, wówczas wynik jest ignorowany

## 2.5 Zakończenie procesu, a kończenie wątków

Kiedy kończy się proces, wszystkie wątki utworzone w jego obrębie są również (gwałtownie) kończone. Dlatego koniecznie trzeba użyć albo wbudowanego mechanizmu synchronizacji, albo jakiegoś własnego rozwiązania.

## 2.6 Rodzaje wątków

W **pthreads** wątki są dwojakiego rodzaju:

- *joinable* (domyślny rodzaj)
- *detached*

Rodzaj jest ustalany w atrybutach wątku. Można jednak po utworzeniu wątku zmienić typ z *joinable* na *detached* funkcją `pthread_detach`; odwrotne działanie **nie jest możliwe**.

Wątek typu *joinable* to taki, z którego można odczytać wartość wynikową zwróconą przez funkcję. Gdy wątek tego typu kończy się, jego zasoby **nie są zwalniane** do chwili wywołania funkcji `pthread_join` (patrz Oczekiwanie na zakończenie wątku).

Warto więc zwrócić uwagę, że utworzenie wątku typu *joinable* i nie wywołanie wspomnianej funkcji skutkować będzie wyciekiem pamięci (następstwem którego tworzenie nowych wątków może stać się niemożliwe).

Wątek typu *detached* z chwilą zakończenia działania od razu zwalnia wszystkie zasoby; funkcja `pthread_join` nie akceptuje identyfikatorów do wątków tego typu.

### 2.6.1 Funkcje

- `int pthread_detach(pthread_t id)`
  - zmiana rodzaju wątku

## 2.7 Przekazywanie argumentów i zwracanie wyników

### 2.7.1 Przekazywanie argumentów

Funkcja wykonywana w wątku przejmuje argument typu `void*` podawany w funkcji `pthread_create` — wskaźnik ten może więc wskazywać dowolną strukturę, może być także pusty.

> **UWAGA**
>
> Ponieważ przekazywane są **wskaźniki**, dla każdego trzeba wątku przypisać osobny obiekt argumentów

Można również rzutować bezpośrednio typy, których rozmiar nie przekracza rozmiaru wskaźnik, tzn. gdy `sizeof(typ) <= sizeof(void*)`; mogą to być typy `int`, `short`, `char`, być może też inne — zależnie od platformy sprzętowej i kompilatora.

### 2.7.2 Zwracanie wyniku

Funkcja użytkownika zwraca wskaźnik na `void*`. Jeśli potrzeba zwrócić jakieś dane, należy je zaalokować na stercie funkcją `malloc` lub podobną.

> **UWAGA**
>
> Zwracanie wskaźników do obiektów lokalnych (utworzonych na stosie) jest błędem — nie tylko w programach wielowątkowych!

Można również rzutować na `void*`, tak samo jak w przypadku przekazywania argumentów.

### 2.7.3 Przykład

⇒ *Przejdź do przykładowego programu nr 2.*

Wyjście:

```
$ ./przyklad
Witaj Wikibooks w dniu 2010-03-14
Wątek 2 wywołany z argumentem liczbowym 27
wątek 2 zwrócił napis: 'xxxxxxxxxxxxxxxxxxxxxxxxxx'
```

## 2.8 Atrybuty wątków

Atrybuty wątku pozwalają ustawić szereg parametrów wątków podczas ich tworzenia. Pojedynczy obiekt atrybutów może być wykorzystany wielokrotnie do tworzenia różnych wątków.

Wszystkie atrybuty wątku opisuje typ `pthread_attr_t`. Nazwy funkcji operujących na tym typie zaczynają się od `pthread_attr`. Funkcje ustawiające poszczególne atrybuty zaczynają się od `pthread_attr_set` i istnieją dla nich odpowiedniki odczytujące `pthread_attr_get`.

## 2.9 Inicjalizacja

Przed użyciem atrybutu zmienna musi zostać zainicjowana funkcją `pthread_attr_init`, zwolnienie zasobów z nim związanych realizuje funkcja `pthread_attr_destroy`.

### 2.9.1 Typy

- `pthread_attr_t`

### 2.9.2 Funkcje

- int pthread_attr_init(pthread_attr_t *attr)
- int pthread_attr_destroy(pthread_attr_t *attr)

### 2.9.3 Przykład

```
pthread_attr_t atrybuty;

pthread_attr_init(&atrybuty);
/* ... */
pthread_attr_destroy(&atrybuty);
```

## 2.10 Rodzaj wątku

Rodzaje wątków zostały dokładniej opisane w innej sekcji. Funkcje `pthread_attr_setdetachstate` ustawia, zaś `pthread_attr_getdetachstate` odczytuje rodzaj wątku, jaki ma zostać ustalony przy jego tworzeniu. Rodzaj jest identyfikowany jedną z wartości:

- `PTHREAD_CREATE_JOINABLE` — utworzenie wątku typu *joinable* (**domyślnie**);
- `PTHREAD_CREATE_DETACHED` — utworzenie wątku typu *detached*.

### 2.10.1 Funkcje

- int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)
  - ustawienie rodzaju
- int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)
  - odczyt

## 2.11 Rozmiar i adres stosu

Domyślny rozmiar stosu wątku zależy od implementacji i może być rzędu kilku-kilkudziesięciu kilobajtów lub kilku megabajtów. Należy liczyć się z tym, że rozmiar będzie wewnętrznie zaokrąglony do rozmiaru strony pamięci (`PAGE_SIZE` — 4kB na procesorach x86). Zmiana rozmiaru jest możliwa jeżeli biblioteka pthreads implementuje opcję *TSS*.

Samodzielne ustalenie rozmiaru stosu może być konieczne, gdy funkcja wątku tworzy duże obiekty na stosie lub przeciwnie — gdy wiadomo, że wątki nie potrzebują zbyt wiele pamięci, a jednocześnie będzie potrzebna dużą ich liczba. Rozmiar stosu nie może być mniejszy od stałej `PTHREAD_STACK_MIN` (z pliku `limits.h`) ani przekraczać możliwości systemu.

Jeśli biblioteka implementuje opcję *TSA* można również ustalić adres stosu.

> **UWAGA**
>
> Zmiana adresu stosu może być nieprzenośne ze względu na to, że wskaźnik stosu może albo zmniejszać albo zwiększać adres przy odkładaniu elementów na stosie

### 2.11.1 Funkcja

- `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)`
  - ustalenie nowego rozmiaru `stacksize`
- `int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize)`
  - odczyt rozmiaru
- int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr)
  - ustalenie nowego rozmiaru stosu `stackaddr`
- int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr)
  - odczyt adresu
- int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize)
  - jednoczesne ustalenie adresu i rozmiaru stosu
- int pthread_attr_getstack(const pthread_attr_t *attr, void **stackaddr, size_t *stacksize)
  - odczyt adresu i rozmiaru

### 2.11.2 Przykład

W programie uruchamiany jest wątek, który na stosie alokuje względnie dużą tablicę (ok. 200kB), następnie tablica jest czyszczona. Program z linii poleceń odczytuje żądany rozmiar stosu — ustawiając jego wartość na zbyt małą z pewnością doprowadzimy do błędu SIGSEGV.

⇒ *Przejdź do przykładowego programu nr 12.*

## 2.12 Obszar zabezpieczający stosu

Opcja *XSI*. Jeśli rozmiar **obszaru zabezpieczającego** (*guard*) jest większy do zera, za stosem wątku rezerwowana jest pamięć (o rozmiarze zaokrąglonym w górę do rozmiaru strony, tj. `PAGE_SIZE`), która nie może być zapisywana ani odczytywana. Ułatwia to detekcję części powszechnych błędów polegających na wyjściu poza stos, czyli np. jego przepełnienie, są bowiem sygnalizowane przez sygnał `SIGSEGV`.

Domyślnie obszar ten jest włączony i ma minimalną wielkość.

### 2.12.1 Funkcje

- `int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize)`
  - ustawienie rozmiaru obszaru zabiezpieczającego
- `int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guardsize)`
  - jego odczyt

## 2.13 Szeregowanie wątków

Wątki, podobnie jak procesy, mogą działać z różnymi priorytetami i być szeregowane przez różne algorytmy. Jest to opcja standardu *TPS*.

### 2.13.1 Dziedzicznie ustawień

Wątek tworzony funkcją `pthread_create` może albo dziedziczyć ustawienia szeregowania z wywołującego wątku, albo uwzględniać wartości z atrybutów. Ten parametr opisują dwie wartości:

- `PTHREAD_INHERIT_SCHED` — ustawienia dziedziczone,
- `PTHREAD_EXPLICIT_SCHED` — ustawienia odczytywane z atrybutów.

#### Funkcje

- `int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched)`
  - ustawienie parametru
- `int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched)`
  - odczytanie parametru

### 2.13.2 Wybór algorytmu szeregowania

Wartość określająca algorytm szeregowania przyjmuje wartości:

- `SCHED_OTHER` (domyślnie),
- `SCHED_FIFO`,
- `SCHED_RR`,
- `SCHED_SPORADIC` (tylko jeśli dostępna jest opcja *TSP*).

> **UWAGA**
>
> Wybór algorytmu może być ograniczony uprawnieniami użytkownika.

#### Funkcje

- `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`
  - ustawienie algorytmu
- `int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`
  - odczyt

### 2.13.3 Wybór algorytmu szeregowania i ustawienie jego parametrów (m.in. priorytet)

Oprócz algorytmu szeregowania można również określić jego parametry, w szczególności priorytet wątku.

> Priorytet wątku jest **niezależny** od priorytetu procesu.

Parametry algorytmu opisuj struktura `sched_param` (z `sched.h`), która zawiera co najmniej jedno pole określające priorytet:

```
struct sched_param {
int sched_priority; /* priorytet */
};
```

Jeśli biblioteka implementuje opcję *TPS* (algorym `SCHED_SPORADIC`), struktura zawiera więcej pól.

Wartość priorytetu jest ograniczona wartościami zwracanymi przez funkcje sched_get_priority_min/max, które zależą od wybranego algorytmu szeregowania.

#### Funkcje

- `int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param)`
  - ustawienie algorytmu i jego parametrów
- `int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param)`
  - odczyt

Info—odczytane parametry nie uwzględniają chwilowych zmian priorytetów, np. spowodowanych ochroną priorytetów w sekcjach krytycznych.

### 2.13.4 Przykład

Program pozwala wybrać algorytm szeregowania, po czym tworzy kilka wątków z co raz większymi priorytetami (z dopuszczalnego zakresu).

⇒ *Przejdź do przykładowego programu nr 19.*

Przykładowe wyjście w systemie Linux dla dostępnych algorytmów szeregowania.

```
$ ./przyklad 0
SCHED_OTHER: priorytety w zakresie 0 ... 0
utworzono wątek #0 o priorytecie 0
utworzono wątek #1 o priorytecie 0
utworzono wątek #2 o priorytecie 0
utworzono wątek #3 o priorytecie 0
wątek #0 (priorytet   0): licznik = 30630
wątek #1 (priorytet   0): licznik = 30631
wątek #2 (priorytet   0): licznik = 30633
wątek #3 (priorytet   0): licznik = 30620

$ ./przyklad 1
SCHED_RR: priorytety w zakresie 1 ... 99
utworzono wątek #0 o priorytecie 1
```

```
utworzono wątek #1 o priorytecie 33
utworzono wątek #2 o priorytecie 66
utworzono wątek #3 o priorytecie 99
wątek #0 (priorytet   1): licznik = 146812
wątek #1 (priorytet  33): licznik = 150084
wątek #2 (priorytet  66): licznik = 151116
wątek #3 (priorytet  99): licznik = 150744

$ ./przyklad 2
SCHED_FIFO: priorytety w zakresie 1 ... 99
utworzono wątek #0 o priorytecie 1
utworzono wątek #1 o priorytecie 33
utworzono wątek #2 o priorytecie 66
utworzono wątek #3 o priorytecie 99
wątek #0 (priorytet   1): licznik = 146659
wątek #1 (priorytet  33): licznik = 149249
wątek #2 (priorytet  66): licznik = 150764
wątek #3 (priorytet  99): licznik = 150895
```

## 2.14 Zakres konkurowania wątków

**Pthreads** pozwala opcjonalnie określić, czy szeregowanie wątków będzie wykonywane w obrębie całego systemu (tzn. ze wszystkimi innymi wątkami i procesami), czy tylko w obrębie wątków z jednego procesu. Jest to opcja standardu *TPS*.

Stałe określające zakres konkurowania:

- `PTHREAD_SCOPE_SYSTEM` — system,
- `PTHREAD_SCOPE_PROCESS` — proces.

**Funkcje**

- `int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)`
  - ustawienie zakresu
- `int pthread_attr_getscope(const pthread_attr_t *attr, int *contentionscope)`
  - odczyt

## 2.15 Przykład

Poniższy program wyświetla wszystkie informacje nt. atrybutów wątku.

⇒ *Przejdź do przykładowego programu nr 4.*

Przykładowe wyjście dla domyślnych ustawień atrybutów (w systemie Linux):

```
$ ./przyklad
atrybuty wątku
* rodzaj: joinable
* adres stosu: (nil)
* rozmiar stosu: 8388608 (minimalny 16384)
* rozmiar obszaru zabezpieczającego: 4096
* parametry szeregowania dziedziczone
* zakres szeregowania: system
```

# Rozdział 3

# Zaawansowane działania

## 3.1 Stos funkcji finalizujących (*cleanup*)

Z każdym wątkiem związany jest **stos funkcji finalizujących** (*cleanup stack*) — jest to osobny stos na którym zapisywane są funkcje użytkownika i argumenty dla nich (przez `pthread_cleanup_push`), które następnie mogą być wywołane wprost przy ściąganiu ze stosu (`pthread_cleanup_pop`).

Ponadto stos funkcji jest **automatycznie** czyszczony — tzn. ściągane i wykonywane są kolejne funkcje przy asynchronicznym przerwaniu wątku oraz gdy wątek jest kończony wywołaniem `pthread_exit`. Jeśli wątek kończy się wykonaniem instrukcji **return**, wówczas to programista jest odpowiedzialny za wyczyszczenie stosu poprzez wywołanie odpowiednią liczbę razy funkcji `pthread_cleanup_pop`. Co prawda standard nakłada na implementację konieczność zagwarantowania, że te dwie funkcje występują w bloku kodu (makra preprocesora), jednak wyjście z bloku instrukcją **return**, **break**, **continue** lub **goto** jest **niezdefiniowane**. (Np. w Cygwinie otrzymałem błąd SIGSEGV, w Linuxie błąd został po cichu zignorowany).

Funkcja użytkownika zwraca i przyjmuje argument typu `void*`.

Zastosowaniem opisanego mechanizmu jest zwalniania zasobów przydzielonych wątkowi — np. zwolnienie blokad, dealokacja pamięci, zamknięcie plików, gniazd. Jest on nieco podobny do znanego z języka C++ automatycznego wołania destruktorów przy opuszczaniu zakresu, w którym zostały utworzone.

### 3.1.1 Funkcje

- int pthread_cleanup_push(void (fun*)(void*), void *arg)
  - odłożenie na stos adresu funkcji `fun`, która przyjmuje argument `arg`
- int pthread_cleanup_pop(int execute)
  - zdjęcie ze stosu funkcji i jeśli `execute` jest różne od zera, wykonanie jej

### 3.1.2 Przykład

W przykładowym programie dwa wątki alokują pamięć. Jeden wątek wprost wywołuje funkcję `pthread_cleanup_pop`, w drugim funkcja finalizująca jest wywoływana automatycznie po wykonaniu `pthread_exit`.

⇒ *Przejdź do przykładowego programu nr 5.*

Wyjście:

```
$ ./przyklad
wątek #0 zaalokował 100 bajtów pod adresem 0x9058098
wątek #1 zaalokował 100 bajtów pod adresem 0x9058190
wątek #0 zaalokował 200 bajtów pod adresem 0x90581f8
wątek #1 zaalokował 200 bajtów pod adresem 0x90582c8
zwalnianie pamięci spod adresu 0x90581f8
zwalnianie pamięci spod adresu 0x9058098
wątek #0 zakończył się
zwalnianie pamięci spod adresu 0x90582c8
zwalnianie pamięci spod adresu 0x9058190
```

## 3.2 Lokalne dane wątku

W **pthreads** istnieje możliwość przyporządkowania **kluczom**, które są jednakowe dla wszystkich wątków, wskaźnika do danych specyficznych dla danego wątku. W istocie jest to powiązanie pary (klucz, wątek) z danymi, przy czym odwołanie do danych wymaga podania jedynie klucza — wywołujący wątek jest domyślnym drugim elementem pary. Ułatwia to m.in. przekazywanie danych do funkcji wywoływanych z poziomu wątków.

Z kluczem można związać funkcję (destruktor), która jest wywoływana przy zakończeniu wątku jeśli dane wątku są różne od `NULL`. Gdy istnieje więcej kluczy, kolejność wywoływania destruktorów jest nieokreślona.

Jeśli klucz został utworzony (`pthread_key_create`) to dane dla nowotworzonego wątku są automatycznie inicjowane na wartość `NULL`. Błędem jest próba sięgnięcia lub zapisania danych dla nieistniejącego klucza.

Liczba dostępnych kluczy jest ograniczona stałą `PTHREAD_KEY_MAX`.

### 3.2.1 Typ

- `pthread_key_t`

### 3.2.2 Funkcje

- int pthread_key_create(pthread_key_t *key, void (destructor*)(void*))
  - utworzenie nowego klucza, przypisanie destruktora
- int pthread_key_delete(pthread_key_t key)
  - usunięcie klucza
- int pthread_setspecific(pthread_key_t key, const void *data)
  - przypisanie do klucza danych wątku
- void* pthread_getspecific(pthread_key_t key)
  - pobranie danych związanych z kluczem.

### 3.2.3 Przykład

W programie tworzony jest jeden klucz, z którym wątki kojarzą napis — przedrostek, którym funkcja `wyswietl` poprzedza wyświetlane komunikaty.

⇒ *Przejdź do przykładowego programu nr 10.*

Przykładowe wyjście:

```
adres napisu: 0x9bd6008 ('***')
adres napisu: 0x9bd60a8 ('!!!')
!!!: Witaj w równoległym świecie!
adres napisu: 0x9bd6148 ('###')
###: Witaj w równoległym świecie!
***: Witaj w równoległym świecie!
!!!: Wątek wykonuje pracę
***: Wątek wykonuje pracę
###: Wątek wykonuje pracę
!!!: Wątek zakończony
wywołano destruktor, adres pamięci do zwolnienia: 0x9bd60a8 ('!!!')
***: Wątek zakończony
wywołano destruktor, adres pamięci do zwolnienia: 0x9bd6008 ('***')
###: Wątek zakończony
wywołano destruktor, adres pamięci do zwolnienia: 0x9bd6148 ('###')
```

## 3.3 Funkcje wywoływane jednokrotnie

Czasem istnieje potrzeba jednokrotnego wykonania jakiejś funkcji, np. w celu inicjalizacji jakiś globalnych ustawień, biblioteki, otwarcia plików, gniazd itp. Pthreads udostępnia funkcję `pthread_once`, która niezależnie od liczby wywołań, uruchamia **dokładnie raz** funkcję użytkownika.

Jednokrotne uruchomienie gwarantuje obiekt typu `pthread_once_t`; zmienną tego typu należy statycznie zainicjować wartością `PTHREAD_ONCE_INIT`.

### 3.3.1 Typy

- `pthread_once_t`

### 3.3.2 Funkcje

- int pthread_once(pthread_once_t *once, void (fun*)(void))

### 3.3.3 Przykład

⇒ *Przejdź do przykładowego programu nr 6.*

Wynik:

```
$ ./przyklad
Rozpoczynanie programu
Uruchomiono wątek nr 0
Uruchomiono wątek nr 2
Uruchomiono wątek nr 1
Uruchomiono wątek nr 3
Uruchomiono wątek nr 4
Uruchomiono wątek nr 5
Uruchomiono wątek nr 6
Uruchomiono wątek nr 7
Uruchomiono wątek nr 8
Uruchomiono wątek nr 9
```

## 3.4 UNIX-owe sygnały

Gdy sygnał zostanie dostarczony do procesu nie jest określone w kontekście którego wątku wykona się procedura obsługi sygnału.

### 3.4.1 Blokowanie sygnałów

Pthreads umożliwia zablokowanie określonych sygnałów na poziomie wątków, służy temu funkcja `pthread_sigmask` (analogiczna do `sigprocmask`), która modyfikuje zbiór zablokowanych sygnałów wywołującego ją wątku. Nie można zablokować `SIGFPE`, `SIGILL`, `SIGSEGV` ani `SIGBUS`

#### Funkcje

- int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset)
  - `how` określa jak zbiór `set` wpływa na bieżący zbiór
    - `SIG_BLOCK` — włączenie wskazanych sygnałów do zbioru
    - `SIG_UNBLOCK` — odblokowanie wskazanych sygnałów
    - `SIG_SETMASK` — zastąpienie bieżącego zbioru nowym

– gdy `oset` nie jest pusty, poprzednia maska sygnałów zapisywana jest pod wskazywanym adresem

### 3.4.2 Wysyłanie sygnałów do wątków

Wysyłanie sygnału do określona wątku umożliwia funkcja `pthread_kill`. Jeśli numer sygnału jest równy zero, wówczas nie jest wysyłany sygnał, ale są testowane jedynie ewentualne błędy — a więc czy wskazany identyfikator wątku jest poprawny.

**UWAGA**

Sygnały, które dotyczą procesów, lecz zostaną wysłane do wątku nie zmieniają swojego znaczenia. Np. wysłanie `SIGSTOP` zatrzyma proces, a nie wątek.

**Funkcje**

- int pthread_kill(pthread_t id, int signum)
  - `id` — wątek
  - `signum` — numer sygnału

### 3.4.3 Przykład

W przykładowym programie wątek główny czeka na sygnał `SIGUSR1`, który po pewnym czasie wysyła utworzony wcześniej wątek.

⇒ *Przejdź do przykładowego programu nr 13.*

Wyjście:

```
$ ./przyklad
wątek główny oczekuje na sygnał
 wątek się rozpoczął
 wątek wysyła sygnał SIGUSR1 do głównego wątku
wątek główny otrzymał sygnał SIGUSR1
```

## 3.5 Przerywanie wątków

Wskazany wątek może zostać przerwany, jeśli tylko nie zostało to wprost zabronione. Sygnał przerwania wysyła funkcja `pthread_cancel`.

Sposób przerwania wątku jest dwojaki:

1. **Asynchroniczny** — przerwanie następuje natychmiast, w dowolnym momencie.
2. **Opóźniony** (*deferred*) — przerwanie następuje dopiero po osiągnięciu tzw. **punktu przerwania** (*cancellation point*), tj. wywołania określonych funkcji systemowych (np. `sleep`, `read`). Standard POSIX określa, które funkcje muszą, a które mogą być punktami przerwania, definiuje także dodatkowo `pthread_testcancel(void)`. Pełna lista funkcji znajduje się w rozdziale [http://www.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_09.html 2.9.5 Thread Cancellation]);

Ustawienie flagi kontrolującej możliwość przerwania wątku wykonuje funkcja `pthread_setcancelstate`, akceptuje dwie wartości:

- `PTHREAD_CANCEL_ENABLE` — przerwanie możliwe;

- `PTHREAD_CANCEL_DISABLE` — przerwanie niemożliwe; jeśli jednak wystąpi żądanie przerwania, **fakt ten jest pamiętany** i gdy stan flagi zmieni się na `PTHREAD_CANCEL_ENABLE` wątek zostanie przerwany.

Wybór sposobu przerywania umożliwia funkcja `pthread_setcanceltype`, która akceptuje dwie wartości:

- `PTHREAD_CANCEL_ASYNCHRONOUS` — przerwanie asynchroniczne,
- `PTHREAD_CANCEL_DEFERRED` — przerwanie opóźnione.

Obie funkcje zmieniają parametry wywołującego je wątku.

### 3.5.1 Funkcje

- int pthread_cancel(pthread_t thread)
  - przerwanie wskazanego wątku
- int pthread_setcancelstate(int state, int *oldstate)
  - ustawia możliwość przerwania i zwraca poprzednią wartość
- int pthread_setcanceltype(int type, int *oldtype)
  - ustawia sposób przerwania i zwraca poprzednią wartość
- void pthread_testcancel(void)
  - punkt przerwania

### 3.5.2 Przykład

Przykładowy program uruchamia trzy wątki z różnymi ustawieniami dotyczącymi przerywania:

1. dopuszcza przerwanie asynchroniczne,
2. dopuszcza przerwanie opóźnione,
3. przez pewien czas w ogóle blokuje przerwania.

⇒ *Przejdź do przykładowego programu nr 7.*

Przykładowe wyjście:

```
$ ./przyklad
 uruchomiono wątek #0 (przerwanie asynchroniczne)
#0: wysyłanie sygnału przerwania do wątku
#0: wysłano, oczekiwanie na zakończenie
 uruchomiono wątek #2 (przez 2 sekundy nie można przerwać)
funkcja finalizująca dla wątku #0
#0: wątek zakończony
#1: wysyłanie sygnału przerwania do wątku
#1: wysłano, oczekiwanie na zakończenie
 uruchomiono wątek #1 (przerwanie opóźnione)
funkcja finalizująca dla wątku #1
#1: wątek zakończony
#2: wysyłanie sygnału przerwania do wątku
#2: wysłano, oczekiwanie na zakończenie
 wątek #2 można już przerwać
funkcja finalizująca dla wątku #2
#2: wątek zakończony
```

## 3.6 Pthreads i forkowanie

Biblioteka zarządza trzema listami funkcji, które są wykonywane przy forkowaniu procesu. Jedna lista przechowuje adresy funkcji wywoływanych **przed** właściwym uruchomieniem funkcji `fork`, dwie kolejne zawierają funkcje wykonywane tuż po zakończeniu `fork`, osobno w procesie rodzica i potomnym.

Funkcja `pthread_atfork` służy do dodawania do list funkcji użytkownika; można podawać puste wskaźniki.

Funkcje wykonywane po forku są wykonywane w kolejności zgodnej z dodawaniem ich do list, natomiast przed forkiem są uruchamiane w **kolejności odwrotnej**.

Zastosowaniem tego mechanizmu może być ponowna inicjalizacja wątków w procesie potomnym. Przede wszystkim przy forkowaniu w procesie potomnym działa tylko jeden wątek — główny wątek, wykonujący funkcję `main`. Ponadto konieczna jest ponowna inicjalizacja różnych obiektów synchronizujących (np. mutexów), bowiem — jak wspomniano we wstępie — samo skopiowanie pamięci jest niewystarczającego do utworzenia w pełni funkcjonalnej kopii takiego obiektu.

### 3.6.1 Funkcje

- int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void))
  - `prepare` — funkcja wykonywana przed `fork()`
  - `parent` — funkcja wykonywana po `fork()` w procesie rodzica
  - `child` — funkcja wykonywana po `fork()` w procesie potomnym

### 3.6.2 Przykład

W programie proces potomny odtwarza jeden działający wątek.

⇒ *Przejdź do przykładowego programu nr 18.*

Wyjście:

```
$ ./przyklad
początek programu
tworzenie 3 wątków w procesie 8228
uruchomiono wątek #1
wątek #1 w procesie #8228
uruchomiono wątek #2
wątek #2 w procesie #8228
uruchomiono wątek #3
wątek #3 w procesie #8228
wątek #1 w procesie #8228
wątek #2 w procesie #8228
wątek #3 w procesie #8228
fork => 8232
tworzenie 3 wątków w procesie 8232
fork => 0
uruchomiono wątek #1
wątek #1 w procesie #8232
uruchomiono wątek #2
wątek #2 w procesie #8232
uruchomiono wątek #3
wątek #3 w procesie #8232
wątek #1 w procesie #8228
wątek #3 w procesie #8228
wątek #2 w procesie #8228
```

```
wątek #1 w procesie #8232
wątek #2 w procesie #8232
wątek #3 w procesie #8232
wątek #1 w procesie #8228
wątek #2 w procesie #8228
wątek #3 w procesie #8228
wątek #1 w procesie #8232
wątek #2 w procesie #8232
wątek #3 w procesie #8232
wątek #1 w procesie #8228
wątek #3 w procesie #8228
wątek #2 w procesie #8228
```

## 3.7 Stopień współbieżności

Dostępne jeśli biblioteka implementuje opcję *XSI*.

Stopień współbieżności jest **podpowiedzią** (*hint*) dla biblioteki i ma znaczenie, jeśli wątki **pthreads** są uruchamiane na mniejszej liczbie wątków systemowych. Wówczas podpowiedź określa na ilu rzeczywistych wątkach zależy programowi.

W Linuxie jeden wątek pthreads odpowiada jednemu wątkowi systemowemu, więc ten parametr nie ma żadnego znaczenia.

### 3.7.1 Funkcja

- int pthread_setconcurrency(int new_level)
  - ustawia nowy stopień współbieżności; jeśli 0, przyjmowany jest domyślny
- int pthread_getconcurrency(void)
  - odczyt

## 3.8 Czas procesora zużyty przez wątek

Jeśli system implementuje timery (*TMR*) i rozszerzenie pthreads (*TCT*) dostępna jest funkcja `pthread_getcpuclockid`, zwracająca identyfikator zegara, który odmierza **czas procesora** zużyty przez wątek. Zdefiniowana jest wówczas także stała w `time.h` `CLOCK_THREAD_CPUTIME_ID`, która odpowiada identyfikatorowi zegara dla wywołującego wątku.

Makrodefinicja `_POSIX_THREAD_CPUTIME` informuje o istnieniu rozszerzenia.

Do odczytania czasu na podstawie identyfikatora zegara służy funkcja `clock_gettime` (z `time.h`).

### 3.8.1 Funkcja

- int pthread_getcpuclockid(pthread_t id, clockid_t *clock_id)
  - odczyt identyfikatora zegara

### 3.8.2 Szkic użycia

```
\begin{enumerate}

\item include <pthread.h>
\item include <time.h>
\end{enumerate}
```

```
pthread_t id_watku;

struct timespec czas; // z time.h
clock_t id_zegara; // z time.h

\begin{enumerate}

\item ifdef _POSIX_THREAD_CPUTIME
\end{enumerate}

if (pthread_getcpuclockid(id_watku, &id_zegara) == 0) {
if (clock_gettime(id_zegara, &czas) == 0) {
printf(
"czas procesora: %ld.%03ld ms\n",
czas.tv_sec,// tv_sec - sekundy
czas.tv_nsec/1000000 // tv_nsec - nanosekundy
);
}
else
/* błąd */
}
else
/* błąd */
\begin{enumerate}

\item else
\item error "pthread_getcpuclockid niedostępne w tym systemie"
\item endif
\end{enumerate}
```

### 3.8.3 Przykład

⇒ *Przejdź do przykładowego programu nr 14.*

Wynik na maszynie dwuprocesorowej:

```
$ ./przyklad
początek programu, uruchomianie zostanie 10 wątków
wątek #0 uruchomiony, dwa razy wykona 86832212 pustych pętli
wątek #7 uruchomiony, dwa razy wykona 8298184 pustych pętli
wątek #9 uruchomiony, dwa razy wykona 67891648 pustych pętli
wątek #5 uruchomiony, dwa razy wykona 27931234 pustych pętli
wątek #8 uruchomiony, dwa razy wykona 23876946 pustych pętli
wątek #3 uruchomiony, dwa razy wykona 52231547 pustych pętli
wątek #6 uruchomiony, dwa razy wykona 16183104 pustych pętli
wątek #4 uruchomiony, dwa razy wykona 87068047 pustych pętli
wątek #1 uruchomiony, dwa razy wykona 59202170 pustych pętli
wątek #2 uruchomiony, dwa razy wykona 48470151 pustych pętli
po około sekundzie wątki zużyły:
* #0: 209ms
* #1: 138ms
* #2: 119ms
* #3: 125ms
```

```
* #4: 209ms
* #5: 67ms
* #6: 41ms
* #7: 17ms
* #8: 59ms
* #9: 154ms
wątek #7 zakończony, zużył 39ms czasu procesora
wątek #6 zakończony, zużył 80ms czasu procesora
wątek #8 zakończony, zużył 117ms czasu procesora
wątek #5 zakończony, zużył 135ms czasu procesora
wątek #2 zakończony, zużył 232ms czasu procesora
wątek #3 zakończony, zużył 251ms czasu procesora
wątek #1 zakończony, zużył 285ms czasu procesora
wątek #9 zakończony, zużył 323ms czasu procesora
wątek #0 zakończony, zużył 423ms czasu procesora
wątek #4 zakończony, zużył 418ms czasu procesora

główny wątek zużył 0ms czasu procesora
proces zużył 2307ms czasu procesora
```

# Rozdział 4

# Synchronizacja

**Pthreads** udostępnia kilka sposobów synchronizacji między wątkami:

- podstawowe
  - Mutexy (blokada na wyłączność)
  - Zmienne warunkowe (*condition variable*)
  - Oczekiwanie na zakończenie wątku
- opcjonalne
  - Blokady zapis/odczyt (*rwlock*)
  - Bariery
  - Wirujące blokady (*spinlock*)

Na tej liście nie ma semaforów, ponieważ zostały zdefiniowane we wcześniejszej wersji standardu POSIX — właściwie istniały wcześniej, jako jeden ze standardowych mechanizmów IPC .

Jeśli biblioteka implementuje opcję *TSH* (Thread Process-Shared Synchronization), wówczas możliwa staje się synchronizacja między wątkami różnych procesów przy użyciu mutexów, zmiennych warunkowych, blokad odczyt/zapis i barier.

## 4.1 Mutexy

**Mutex** (***MUT****ual* ***EX****clusion*, wzajemne wykluczanie) jest blokadą, którą może uzyskać **tylko jeden wątek**. Mutexy służą głównie do realizacji sekcji krytycznych , czyli bezpiecznego w sensie wielowątkowym dostępu do zasobów współdzielonych.

Schemat działania na mutexach jest następujący:

1. pozyskanie blokady
2. modyfikacja lub odczyt współdzielonego obiektu
3. zwolnienie blokady

Mutex w **pthreads** jest opisywany przez strukturę typu `pthread_mutex_t`, zaś jego atrybuty `pthread_mutexattr_t`.

### 4.1.1 Inicjalizacja i zwalnianie mutexu

Zmienna typu `pthread_mutex_t` może zostać zainicjowana na dwa sposoby:

- poprzez przypisanie symbolu `PTHREAD_MUTEX_INITIALIZER`;
- przez wywołanie funkcji `pthread_mutex_init`, która umożliwia również podanie atrybutów blokady.

Każdy mutex, **bez względu na sposób inicjalizacji**, musi zostać zwolniony funkcja `pthread_mutex_destroy`. Implementacja biblioteki może bowiem `PTHREAD_MUTEX_INITIALIZER` realizować poprzez wywołanie jakiejś funkcji, która np. alokuje pamięć i nie zwolnienie mutexu doprowadzi do wycieku pamięci.

#### Typy

- `pthread_mutex_t`
  - mutex

#### Funkcje

- int pthread_mutex_create(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
  - inicjacja mutexa, wskaźnik na atrybuty `attr` może być pusty
- int pthread_mutex_destroy(pthread_mutex_t *mutex)
  - zwolenienie mutexu

#### Przykład

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* lub */

pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr); /* inicjalizacja atrybutów mutexa */
pthread_mutex_init(&mutex, &attr); /* inicjalizacja mutexa */
...
pthread_mutex_destroy(&mutex); /* zwolnienie zasobów związanych z mutexem */
```

### 4.1.2 Pozyskiwanie i zwolnienie blokady

Pozyskanie blokady umożliwiają trzy funkcje:

1. `pthread_mutex_lock`,
2. `pthread_mutex_trylock`,
3. `pthread_mutex_timedlock`.

Jeśli żaden inny wątek nie posiada blokady, działają identycznie — tzn. blokada jest przyznawana wywołującemu wątkowi. Różnią się zachowaniem w przypadku niemożności uzyskania blokady:

1. `pthread_mutex_lock` — oczekiwanie w nieskończoność, aż blokada zostanie zwolniona przez inny wątek;
2. `pthread_mutex_trylock` — sterowanie wraca natychmiast, zwracając kod `EBUSY`;
3. `pthread_mutex_timedlock` — oczekiwanie ograniczone czasowo, jeśli czas minie, zwraca kod `ETIMEDOUT`.

Wątek musi **zwolnić** blokadę funkcją `pthread_unlock`.

Funkcja `pthread_mutex_timedlock` jest dostępna, gdy system implementuje rozszerzenie *TMO*. W odróżnieniu od innych funkcji operujących na czasach oczekiwania (np. `select` dla plików), w których podaje się ile czasu ma upłynąć od chwili wywołania funkcji, w **pthreads** podawany jest **czas bezwzględny**.

#### Funkcje

- int pthread_mutex_lock(pthread_mutex_t *mutex)
- int pthread_trylock(pthread_mutex_t *mutex)
- int pthread_timedlock(pthread_mutex_t *mutex, const struct timespec *timeout)

#### Szkic użycia

```
#include <pthread.h>
#include <errno.h>
#include <time.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* ... */
pthread_mutex_lock(&mutex);
// działania na obiekcie współdzielonym
pthread_mutex_unlock(&mutex);

/* ... */
switch (pthread_mutex_try_lock(&mutex)) {
case 0:
// działania na obiekcie współdzielonym
pthread_mutex_unlock(&mutex);
break;

case EBUSY:
puts("Blokadę posiada inny wątek");
break;

default:
```

```
puts("Inny błąd");
break
}

/* ... */
struct timespec timeout;
clock_gettime(CLOCK_REALTIME, &timeout); // pobranie bieżącego czasu
timeout.tv_sec += 2; // zwiększenie liczby sekund o 2

switch (pthread_mutex_timedlock(&mutex, &timeout)) {
case 0:
puts("Blokada pozyskana przed upływem 2 sekund");
// działania na obiekcie współdzielonym
pthread_mutex_unlock(&mutex);
break;

case ETIMEDOUT:
puts("Upłynęły 2 sekundy");
break;

default:
puts("Inny błąd");
break;
}
```

**Przykład**

Program demonstruje sekcję krytyczną z użyciem mutexów. Jeśli przy kompilacji zdefiniowane zostanie `BLOKADA`, wówczas mutex blokuje dostęp do zmiennej, która jest inkrementowana określoną liczbę razy przez każdy z wątków. W przeciwnym razie wątki zmieniają ją bez żadnej synchronizacji, co może prowadzić do błędu — w tym przypadku do niepoprawnego zliczenia.

⇒ *Przejdź do przykładowego programu nr 15.*

Przykładowe wyjście (z błędem):

```
$ ./przyklad
licznik = 0
licznik = 9968, spodziewana wartość = 10000 BŁĄD!!!
```

### 4.1.3 Typy mutexów

Opcja *XSI*. Jednym z atrybutów mutexu jest jego typ, czy też rodzaj:

1. zwykły (*normal*),
2. rekursywny (*recursive*),
3. bezpieczny (*error check*).

Na poziomie współpracy między wątkami rodzaj mutexu nie ma znaczenia, objawia się dopiero w obrębie jednego wątku w dwóch sytuacjach:

- ponowna próba pozyskania blokady,
- próba zwolnienia już zwolnionej blokady.

### Ponowne blokowanie

Można wyobrazić sobie sytuację (raczej prawdopodobną), gdy w programie istnieje funkcja pomocnicza, wykorzystywana przez wątki, która zakłada blokadę na pewne dane. Problem pojawia się w chwili, gdy wątek już pozyskał blokadę i wywołuje taką funkcję. Wówczas z punktu widzenia blokady wątek próbuje wykonać następującą sekwencję:

```
pthread_mutex_lock(&mutex); // (1)
pthread_mutex_lock(&mutex); // (2) - w funkcji pomocniczej
/* ... */
pthread_mutex_unlock(&mutex) // (3) - w funkcji pomocniczej
pthread_mutex_unlock(&mutex) // (4)
```

- W przypadku mutexu **zwykłego** wykona się pierwsza funkcja `pthread_mutex_lock` (1), zaś na drugim jej wywołaniu (2) wątek zatrzyma się, oczekując na zwolnienie blokady — co **nigdy nie nastąpi**, bowiem sterowanie nie dojdzie do wiersza (3) ani (4). Występuje **zakleszczenie**.
- W przypadku mutexu **rekursywnego** wykonają się wszystkie funkcje związane z blokadą. Mutex tego typu posiada dodatkowy licznik zagnieżdżeń, który z każdym wywołaniem funkcji `pthrad_mutex_lock` jest zwiększany, natomiast wywołanie `pthread_mutex_unlock` zmniejsza go — gdy osiągnie zero, blokada jest zwalniana.
- W przypadku mutexu **bezpiecznego** drugie wywołanie `pthread_mutex_lock` zwróci kod błędu `EDEADLK`, oznaczający, że wątek już posiada tę blokadę.

### Ponowne odblokowanie

Jeśli blokada jest zwolniona ponowne wywołanie `pthread_unlock` mutexy **bezpieczne** i **rekursywny** zwracają błąd. Zachowanie **zwykłego** mutexu jest **nieokreślone**!

### Przykład

Ilustracja różnicy w działaniu mutexów.

⇒ *Przejdź do przykładowego programu nr 8.*

- Wynik dla mutexu zwykłego — wystąpiło zakleszczenie, program "zawiesił się" i musiał zostać przerwany ręcznie:

```
$ ./przyklad 0
mutex typu PTHREAD_MUTEX_NORMAL
przed wykonaniem pthread_mutex_lock (1)
... wykonano pthread_mutex_lock (1)
przed wykonaniem pthread_mutex_lock (2)
\textbf{^C}
```

- Wynik dla mutexu sprawdzającego — nie dopuszczono do zakleszczenia:

```
$ ./przyklad 1
mutex typu PTHREAD_MUTEX_ERRORCHECK
przed wykonaniem pthread_mutex_lock (1)
... wykonano pthread_mutex_lock (1)
przed wykonaniem pthread_mutex_lock (2)
\textbf{pthread_mutex_lock (2): Resource deadlock avoided}
```

- Wynik dla mutexu rekursywnego — blokada jest pozyskiwana wielokrotnie:

```
$ ./przyklad 2
mutex typu PTHREAD_MUTEX_RECURSIVE
przed wykonaniem pthread_mutex_lock (1)
... wykonano pthread_mutex_lock (1)
przed wykonaniem pthread_mutex_lock (2)
... wykonano pthread_mutex_lock (2)
przed wykonaniem pthread_mutex_unlock (2)
... wykonano pthread_mutex_unlock (2)
przed wykonaniem pthread_mutex_unlock (1)
... wykonano pthread_mutex_unlock (1)
program zakończony
```

### 4.1.4 Atrybuty mutexu

#### Inicjalizacja i usuwanie

#### Typy

- `pthread_mutexattr_t`

#### Funkcje

- int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
- int pthread_mutexattr_init(pthread_mutexattr_t *attr)

#### Typ mutexu

Opisane wyżej

#### Funkcje

- int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
- int pthread_mutexattr_gettype(const pthread_mutexattr_t *atter, int *type)

#### Współdzielenie mutexu z innymi procesami

Patrz rozdział synchronizacja między wątkami różnych procesów.

#### Funkcje

- int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared)
- int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *pshared)

#### Zmiana priorytetu wątku posiadającego blokadę

Dostępne, gdy istnieje rozszerzenie *TPP* (oraz *TPI*).

Wartość atrybutu decyduje o strategii wykonywania programu, gdy wiele wątków o różnych priorytetach stara się o uzyskanie blokady. Atrybut może mieć wartości:

1. `PTHREAD_PRIO_NONE`,
2. `PTHREAD_PRIO_PROTECT`,
3. `PTHREAD_PRIO_INHERIT` (opcja *TPI*).

W przypadku `PTHREAD_PRIO_NONE` priorytet wątku, który pozyskuje blokadę nie zmienia.

W dwóch pozostałych przypadkach z mutexem powiązany zostaje pewien priorytet i gdy wątek uzyska blokadę, wówczas jego priorytet jest podbijany do wartość z mutexu (o ile oczywiście był wcześniej niższy). Innymi słowy w obrębie sekcji krytycznej wątek może działać z wyższym priorytetem.

Sposób ustalania priorytetu mutexu zależy od atrybutu:

- `PTHREAD_PRIO_INHERIT` — wybierany jest maksymalny priorytet spośród wątków oczekujących na uzyskanie danej blokady;
- `PTHREAD_PRIO_PROTECT` — priorytet jest ustalany przez programistę funkcją `pthread_mutexattr_setprioceiling` lub `pthread_mutex_setprioceiling` (opisane w następnej sekcji).

Dodatkowo jeśli wybrano wartość `PTHREAD_PRIO_PROTECT`, wówczas wszelkie próby założenia blokady funkcjami `pthread_mutex_XXXlock` z poziomu wątków o priorytecie niższym niż ustawiony dla mutexa nie powiodą się — zostanie zwrócona wartość błędu `EINVAL`.

#### Funkcje

- int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)
- int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol)

#### Minimalny priorytet wątku zakładające blokadę

Dostępne w opcji *TPP*. Funkcje ustalają/odczytują bieżący priorytet związany z mutexem.

#### Funkcje działające na atrybutach

- int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling)
- int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int *prioceiling)

#### Funkcje działające bezpośrednio na mutexie

- int pthread_mutex_setprioceiling(pthread_mutex_t *attr, int prioceiling)
- int pthread_mutex_getprioceiling(const pthread_mutex_t *attr, int *prioceiling)

## 4.2 Zmienne warunkowe

**Zmienna warunkowa** (*condition variable*) jest jednym ze sposobów synchronizacji między wątkami — polega na przesłaniu sygnału z jednego wątku do innych wątków, które na ten sygnał oczekują.

Prościej rzecz ujmując jeden lub kilka wątków może oczekiwać na zajście jakiegoś warunku, inny wątek, gdy go spełni, sygnalizuje właśnie poprzez **zmienną warunkową** ten fakt jednemu lub wszystkim oczekującym. We wzorcu producent-konsument występuje właśnie taka sytuacja: konsument (jeden lub więcej) czeka na pojawienie się obiektów od producenta (jednego lub więcej).

Zmienna warunkowa jest **zawsze używana z mutexem**.

Typem danych, który opisuje zmienną warunkową jest `pthread_cond_t`.

## 4.3 Schemat użycia zmiennej warunkowej podczas oczekiwania

1. pozyskaj blokadę (mutex)
2. sprawdź, czy warunek zaszedł
3. jeśli tak, zwolnij blokadę
4. w przeciwnym razie oczekuj na sygnał (w tym miejscu blokada jest **automatycznie zwalniana**)
5. * sygnał nadszedł, przejdź do punktu 2 (w tym miejscu blokada jest **automatycznie ponownie pozyskiwana**)

Co w języku C wygląda mniej więcej tak:

```
pthread_mutex_lock(&mutex)
do {
if (warunek spełniony) {
/* ... */
break;
}
else
pthread_cond_wait(&cond, &mutex);
} while (1)
pthread_mutex_unlock(&mutex);
```

## 4.4 Inicjalizacja zmiennej warunkowej

Funkcją `pthread_cond_init` inicjuje zmienną warunkową, umożliwia również przypisanie atrybutów. Istnieje możliwość statycznej inicjalizacji wartością `PTHREAD_COND_INITIALIZER`.

Każda zmienna warunkowa, bez względu na sposób inicjalizacji, musi zostać zwolniony funkcja `pthread_cond_destroy`. Implementacja biblioteki może bowiem `PTHREAD_COND_INITIALIZER` realizować poprzez wywołanie jakiejś funkcji, która np. alokuje pamięć i nie zwolnienie zmiennej doprowadzi do wycieku pamięci.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_cond_attr_t attr;

/* ... */
pthread_cond_init(&cond, NULL); /* inicjalizacja zmiennej warunkowej z atrybutami domyślnymi */

/* ... */
pthread_condattr_init(&attr); /* inicjalizacja atrybutów zmiennej */
pthread_cond_init(&cond, &attr); /* inicjalizacja zmiennej warunkowej z atrybutami użytkownika */
```

```
/* ... */
pthread_cond_destroy(&cond); /* skasowanie zmiennej warunkowej */
pthread_condattr_destroy(&attr); /* oraz atrybutów */
```

### 4.4.1 Typy

- `pthread_cond_t`
  - zmienna warunkowa
- `pthread_condattr_t`
  - atrybuty zmiennej warunkowej

### 4.4.2 Funkcje

- int pthread(pthread_cond_t *cond, cond pthread_condattr_t *attr)
  - inicjacja zmiennej warunkowej, wskaźnik na atrybuty `attr` może być pusty
- int pthread(pthread_cond_t *cond)
  - zwolenie zmiennej warunkowej

## 4.5 Atrybuty

Zmienna warunkowa może mieć dwa atrybuty, zależnie od rozszerzeń:

- *THS* — czy zmienna warunkowa może być współdzielona między procesami (domyślnie nie);
- *CT* — ustawienie identyfikatora zegara, który jest używany przy odmierzaniu czasu w funkcji `pthread_cond_timedwait`

### 4.5.1 Typy

- `pthread_condattr_t`
  - atrybuty zmiennej warunkowej

### 4.5.2 Funkcje

- int pthread_condattr_init(pthread_condattr_t *attr)
  - inicjacja atrybutów zmiennej warunkowej
- int pthread_condattr_destroy(pthread_condattr_t *attr)
  - zwolenienie atrybutów zmiennej warunkowej
- int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared)
  - ustawienie flagi współdzielenia z innymi procesami; `pshared` ma wartość `PTHREAD_PROCESS_SHARED` lub `PTHREAD_PROCESS_PRIVATE` (domyślnie)
- int pthread_condattr_getpshared(pthread_condattr_t *attr, int *pshared)
  - odczyt flagi
- int pthread_condattr_setclock(pthread_condattr_t *attr, clockid_t clock_id)

  - ustawienie identyfikatora zegara

- int pthread_condattr_getclock(pthread_condattr_t *attr, clockid_t *clock_id)
  - odczyt identyfikatora

## 4.6 Sygnalizowanie i rozgłaszanie

Wątek, który zmienił warunek informuje o tym fakcie oczekujące wątki na dwa sposoby:

- sygnalizując (`pthread_cond_signal`) — sygnał trafia do **jednego** z oczekujących wątków, przy czym nie jest określone do którego;
- rozgłaszając (`pthread_cond_broadcast`) — sygnał trafia do **wszystkich** oczekujących wątków.

### 4.6.1 Funkcje

- int pthread_cond_signal(pthread_cond_t *cond)
- int pthread_cond_broadcast(pthread_cond_t *cond)

## 4.7 Oczekiwanie

Oczekiwanie na zmienną warunkową umożliwiają dwie funkcje:

1. `pthread_cond_wait` — oczekuje w nieskończoność na sygnał,
2. `pthread_cond_timedwait` — oczekuje przez ograniczony czas.

Sposób obsługi czasu w **pthreads** został opisany dokładnie przy okazji `pthread_mutex_timedwait` (Pozyskiwanie i zwolnienie blokady).

### 4.7.1 Funkcje

- 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)

## 4.8 Przykład

⇒ *Przejdź do przykładowego programu nr 9.*

Przykładowe wyjście:

```
$ ./przyklad
początek programu
 uruchomiono wątek #5
 wątek #5 oczekuje na sygnał...
 uruchomiono wątek #3
 wątek #3 oczekuje na sygnał...
 uruchomiono wątek #1
 wątek #1 oczekuje na sygnał...
 uruchomiono wątek #4
 wątek #4 oczekuje na sygnał...
 uruchomiono wątek #2
```

```
 wątek #2 oczekuje na sygnał...
pthread_cond_signal - sygnalizacja
 ... wątek #5 otrzymał sygnał!
 wątek #5 oczekuje na sygnał...
pthread_cond_broadcast - rozgłaszanie
 ... wątek #3 otrzymał sygnał!
 wątek #3 oczekuje na sygnał...
 ... wątek #1 otrzymał sygnał!
 wątek #1 oczekuje na sygnał...
 ... wątek #4 otrzymał sygnał!
 wątek #4 oczekuje na sygnał...
 ... wątek #2 otrzymał sygnał!
 wątek #2 oczekuje na sygnał...
 ... wątek #5 otrzymał sygnał!
 wątek #5 oczekuje na sygnał...
koniec programu
```

## 4.9 Wstęp

Mutexy czy wirujące blokady umożliwiają ochronę współdzielonych zasobów jednakowo traktując wątki zmieniające ten obiekt jak i wątki jedynie czytające — dopuszczając do obiektu tylko jeden wątek.

Blokady zapis/odczyt (*rwlocks*) rozróżniają cel dostępu do obiektu współdzielonego — wątek może założyć blokadę do odczytu (*rdlock*) lub do zapisu (*wrlock*). **Dowolna liczba** wątków może mieć jednoczesny dostęp do obiektu chronionego, jeśli tylko zakładają blokadę do odczytu, natomiast **dokładnie jeden wątek** ma dostęp do obiektu, gdy założy blokadę do zapisu.

## 4.10 Inicjalizacja i destrukcja

Blokadę tworzy funkcja `pthread_rwlock_init`, natomiast niszczy `pthread_rwlock_destroy`.

### 4.10.1 Typy

- `pthread_rwlock_t`
- `pthread_rwlockattr_t`

### 4.10.2 Funkcje

- pthread_rwlock_init
- pthread_rwlock_destroy

## 4.11 Atrybuty blokady

Atrybuty blokady tworzy funkcja `pthread_rwlockattr_init`, natomiast niszczy `pthread_rwlockattr_destroy`.

Jeśli biblioteka implementuje opcję *THS*, wówczas blokada ma tylko jeden parametr: flagę współdzielenia między procesami, którą ustawia `pthread_rwlockattr_setpshared`, zaś odczytuje `pthread_rwlockattr_getpshared`.

### 4.11.1 Typy

- `pthread_rwlockattr_t`

### 4.11.2 Funkcje

- pthread_rwlockattr_init(pthread_rwlockattr_t *attr)
  - incjalizacja
- pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr)
  - zniszczenie
- pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared)
  - ustawienie flagi, `pshared` ma wartość `PTHREAD_PROCESS_PRIVATE` lub `PTHREAD_PROCESS_SHARED`
- pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int pshared
  - odczyt flagi

## 4.12 Blokady do odczytu

### 4.12.1 Funkcje

- pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
  - oczekiwanie na pozyskanie blokady w nieskończoność
- pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock)
  - bez oczekiwania na pozyskanie blokady
- pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock, const struct timespec *abs_timeout)
  - oczekiwanie na pozyskanie blokady ograniczone czasowo (dostępne jeśli biblioteka implementuje opcję *TMO*)

## 4.13 Blokady do zapisu

### 4.13.1 Funkcje

- pthread_rwlock_wrlock(pthread_rwlock_t *rwlock)
  - oczekiwanie na pozyskanie blokady w nieskończoność
- pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock)
  - bez oczekiwania na pozyskanie blokady
- pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock, const struct timespec *abs_timeout)
  - oczekiwanie na pozyskanie blokady ograniczone czasowo (dostępne jeśli biblioteka implementuje opcję *TMO*)

## 4.14 Zwalnianie blokady

Niezależnie od rodzaju pozyskanej blokady, wątek zwalnia ją funkcją `pthread_rwlock_unlock`.

### 4.14.1 Funkcje

- int pthread_rwlock_unlock(pthread_rwlock_t *rwlock)

## 4.15 Przykład

Przykładowy program tworzy kilka wątków piszących, a więc zakładających blokady do zapisu, oraz więcej wątków czytających, zakładających blokady do odczytu.

⇒ *Przejdź do przykładowego programu nr 16.*

Przykładowy wynik:

```
$ ./przyklad
pisarz #0 czeka na dostęp
pisarz #0 ustawia nową wartość
pisarz #1 czeka na dostęp
 czytelnik #0 czeka na dostęp
 czytelnik #1 czeka na dostęp
 czytelnik #3 czeka na dostęp
```

```
 czytelnik #2 czeka na dostęp
 czytelnik #4 czeka na dostęp
pisarz #0 zwalnia blokadę
pisarz #1 ustawia nową wartość
pisarz #1 zwalnia blokadę
 czytelnik #0 odczytuje wartość
 czytelnik #4 odczytuje wartość
 czytelnik #1 odczytuje wartość
 czytelnik #3 odczytuje wartość
 czytelnik #2 odczytuje wartość
 czytelnik #4 zwalnia blokadę
 czytelnik #1 zwalnia blokadę
 czytelnik #2 zwalnia blokadę
 czytelnik #0 zwalnia blokadę
 czytelnik #3 zwalnia blokadę
 czytelnik #3 czeka na dostęp
 czytelnik #3 odczytuje wartość
 czytelnik #0 czeka na dostęp
 czytelnik #0 odczytuje wartość
 czytelnik #2 czeka na dostęp
 czytelnik #2 odczytuje wartość
 czytelnik #1 czeka na dostęp
 czytelnik #1 odczytuje wartość
 czytelnik #4 czeka na dostęp
 czytelnik #4 odczytuje wartość
 czytelnik #2 zwalnia blokadę
 czytelnik #3 zwalnia blokadę
 czytelnik #4 zwalnia blokadę
 czytelnik #0 zwalnia blokadę
 czytelnik #1 zwalnia blokadę
pisarz #0 czeka na dostęp
pisarz #0 ustawia nową wartość
 czytelnik #2 czeka na dostęp
 czytelnik #3 czeka na dostęp
 czytelnik #4 czeka na dostęp
 czytelnik #1 czeka na dostęp
 czytelnik #0 czeka na dostęp
pisarz #0 zwalnia blokadę
 czytelnik #2 odczytuje wartość
 czytelnik #4 odczytuje wartość
pisarz #1 czeka na dostęp
 czytelnik #0 odczytuje wartość
 czytelnik #3 odczytuje wartość
 czytelnik #1 odczytuje wartość
 czytelnik #1 zwalnia blokadę
 czytelnik #3 zwalnia blokadę
 czytelnik #0 zwalnia blokadę
 czytelnik #2 zwalnia blokadę
 czytelnik #4 zwalnia blokadę
pisarz #1 ustawia nową wartość
```

## 4.16 Wstęp

**Bariera** jest mechanizmem synchronizacji grupy wątków. Wątki po dojściu bo bariery są wstrzymywane do czasu, aż ostatni jej nie osiągnie — wówczas wszystkie są kontynuowane. Z barierą związana jest liczba większa od zera określająca ile wątków wchodzi w skład grupy.

Gdy bariera jest osiągana przez wszystkie wątki, jej stan (licznik) jest automatycznie inicjowany, na taką wartość, jaką ustawiło ostatnie wywołanie `pthread_barrier_init`.

## 4.17 Inicjalizacja bariery

Inicjalizację bariery wykonuje funkcja `pthread_barrier_init`, usuwa zaś funkcja `pthread_barrier_destroy`. Bariera może mieć dodatkowe atrybut.

### 4.17.1 Typy

- `pthread_barrier_t`
  - bariera
- `pthread_barrierattr_t`
  - atrybuty bariery

### 4.17.2 Funkcje

- `int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t* attr, unsigned int count)`
  - inicjacja bariery, `count` jest liczbą wątków w grupie, `attr` opcjonalne atrybuty
- `int pthread_barrier_destroy(pthread_barrier_t *barrier)`
  - usunięcie bariery

## 4.18 Atrybuty bariery

Gdy biblioteka implementuje opcję *TSH*, wówczas można ustalić, czy bariery mogą być współdzielone między procesami — domyślnie nie.

### 4.18.1 Funkcje

- `int pthread_barrier_init(pthread_barrierattr_t *attr)`
  - inicjacja atrybutów bariery
- `int pthread_barrierattr_destroy(pthread_barrierattr_t *attr)`
  - usunięcie atrybutów
- `int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared)`
  - ustawienie flagi współdzielenia z innymi procesami; `pshared` ma wartość `PTHREAD_PROCESS_SHARED` lub `PTHREAD_PROCESS_PRIVATE` (domyślnie)
- `int pthread_barrierattr_getpshared(pthread_barrierattr_t *attr, int *pshared)`
  - odczytanie flagi

## 4.19 Bariera

Wywołanie funkcji `pthread_barrier_wait` jest traktowane jako dojście do bariery — powoduje zwiększenie licznika związanego z barierą.

Gdy bariera zostanie osiągnięta przez wszystkie wątki, funkcja w jednym wątku (standard nie określa w którym) zwraca specjalną wartość `PTHREAD_BARRIER_SERIAL_THREAD`, pozostałe wartość 0.

### 4.19.1 Funkcje

- `int pthread_barrier_wait(pthread_barrier_t *barrier)`

### 4.19.2 Przykład

W programie bariera służy do wstrzymania programu, do czasu aż wszystkie utworzone wątki skończą działanie.

⇒ *Przejdź do przykładowego programu nr 11.*

Przykładowe wyjście:

```
zostanie uruchomionych 10 wątków
 wątek #0 rozpoczęty, zostanie wstrzymany na 0 sekund
 wątek #0 osiągnął barierę
 wątek #1 rozpoczęty, zostanie wstrzymany na 0 sekund
 wątek #1 osiągnął barierę
 wątek #2 rozpoczęty, zostanie wstrzymany na 0 sekund
 wątek #2 osiągnął barierę
 wątek #3 rozpoczęty, zostanie wstrzymany na 0 sekund
 wątek #3 osiągnął barierę
 wątek #4 rozpoczęty, zostanie wstrzymany na 4 sekund
 wątek #5 rozpoczęty, zostanie wstrzymany na 4 sekund
 wątek #6 rozpoczęty, zostanie wstrzymany na 3 sekund
 wątek #7 rozpoczęty, zostanie wstrzymany na 4 sekund
 wątek #8 rozpoczęty, zostanie wstrzymany na 4 sekund
 wątek #9 rozpoczęty, zostanie wstrzymany na 4 sekund
wątek główny osiągnął barierę
 wątek #6 osiągnął barierę
 wątek #4 osiągnął barierę
 wątek #5 osiągnął barierę
 wątek #7 osiągnął barierę
 wątek #8 osiągnął barierę
 wszystkie wątki osiągnęły barierę (wątek #9)
```

## 4.20 Wstęp

Wirująca blokada (*spinlock*) jest rodzajem blokady, w której **aktywnie** oczekuje się na jej zwolnienie w przypadku, gdy nie może zostać pozyskana. Pod względem semantyki nie różnią się od mutexów, jednak zasadniczo różnią w implementacji i zakresie zastosowania.

Przede wszystkim *spinlock* zużywa czas procesora, więc jego stosowanie ma sens, gdy współdzielone zasoby nie są zbyt długo blokowane przez inne wątki. Ponadto na systemach wieloprocesorowych unika się kosztownego przełączania kontekstu.

## 4.21 Inicjalizacja i zwalnianie

### 4.21.1 Typy

- `pthread_spinlock_t`

### 4.21.2 Funkcje

- `int pthread_spin_init(pthread_spinlock_t *lock, int pshared)`
  - inicjalizacja wirującej blokady; argument `pshared` przymuje wartości `PTHREAD_PROCESS_PRIVATE` lub `PTHREAD_PROCESS_SHARED`, jeśli dostępna jest opcja *THS* — patrz synchronizacja między wątkami różnych procesów
- `int pthread_spin_destroy(pthread_spinlock_t *lock)`
  - zniszczenie wirującej blokady

## 4.22 Pozyskiwanie blokady

### 4.22.1 Funkcje

- `int pthread_spin_lock(pthread_spinlock_t *lock)`
  - założenie blokady, jeśli nie jest to możliwe — oczekiwanie
- `int pthread_spin_trylock(pthread_spinlock_t *lock)`
  - założenie blokady, jeśli nie jest to możliwe — zwrócenie kodu błędu `EBUSY`

## 4.23 Zwalnianie blokady

### 4.23.1 Funkcje

- `int pthread_spin_unlock(pthread_spinlock_t *lock)`
  - zwolnienie blokady

Jeśli biblioteka **pthreads** implementuje opcję *TSH* (Thread Process-Shared Synchronization), wówczas możliwa staje się synchronizacja między wątkami różnych procesów przy użyciu:

- mutexów,
- zmiennych warunkowych,
- blokad odczyt/zapis,
- barier.

Domyślnie obiekty synchronizujące są lokalne względem procesu, w ramach którego zostały utworzone. Jeśli są współdzielone z innymi procesami, wówczas można używać adresów obiektów znajdujących się w segmencie pamięci dzielonej.

> **UWAGA**
>
> Standard nie określa, co się stanie jeśli prywatne obiekty synchronizujące zostaną użyte przez wątki innego procesu. Np. w Linuxie nie są zgłaszane żadne błędy, lecz synchronizacja nie funkcjonuje.

To, czy obiekt synchronizujący będzie współdzielony decydują jego atrybuty — ustawiane funkcjami `int pthread_XXXattr_setpshared(pthread_XXXattr_t *attr, int shared)` (XXX=mutex, cond, rwlock, barrier), gdzie parametr `shared` przyjmuje wartości:

- `PTHREAD_PROCESS_PRIVATE` (domyślnie) — obiekt prywatny;
- `PTHREAD_PROCESS_SHARED` — współdzielony.

## 4.24 Przykład

Przykładowy program w zależności od argumentów:

1. Tworzy segment pamięci dzielonej, podłącza go do przestrzeni adresowej procesu, w którego obszarze inicjalizuje mutex i zmienną warunkową na współdzielone (`PTHREAD_PROCESS_SHARED`) i oczekuje na warunek — ustawienie napisu.
2. Podłącza się do już utworzonego segmentu pamięci, ustawia napis i sygnalizuje zmienną warunkową ten fakt.

⇒ *Przejdź do przykładowego programu nr 17.*

```
$ ./przyklad
proces 1
id segmentu pamięci dzielonej: 27197465
adres przyłączonego segmentu: 0xb76e7000
pthread_cond_wait




inny proces ustawił napis: 'czy konie mnie słyszą?'
$
```

```
$ ./przyklad 27197465 "czy konie mnie słyszą?"
proces 2: segment pamięci dzielonej 27197465
adres przyłączonego segmentu: 0xb7727000
proces ustawił napis 'czy konie mnie słyszą?' i wykonał pthread_cond_signal
$
```

# Rozdział 5

# Rozszerzenia

## 5.1 Linux

Sufiksem nazw większości funkcji Linuxa jest `_np`.

### 5.1.1 Zbiór procesorów, na jakich może uruchomić się wątek

Funkcje umożliwiają ustawienie i odczyt zbioru procesorów na jakich wątek ma działać. (Analogiczne ustawienia są możliwe na poziomie procesów).

- `pthread_setaffinity_np (3)`
- `pthread_getaffinity_np (3)`
- `pthread_attr_getaffinity_np (3)`
- `pthread_attr_setaffinity_np (3)`

### 5.1.2 Funkcje finalizujące i asynchroniczne przerwania

- pthread_cleanup_push_defer_np (3) — funkcja działa podobnie, jak `pthread_cleanup_push`, z tym że po odłożeniu funkcji na stos ustawia sposób przerwania wątku na opóźniony (jednocześnie zapamiętując bieżące ustawiania)
- pthread_cleanup_pop_restore_np (3) — ściąga za stosu funkcję i ewentulanie uruchamia, odtwarza poprzedni sposób przerywania

### 5.1.3 Bieżące atrybuty wątku

Umożliwia odczytanie bieżących atrybutów już uruchomionego wątku.

- pthread_getattr_np (3)

### 5.1.4 Oczekiwania na zakończenie wątku

Uzupełnienie mechanizmu oczekiwania na zakończenie wątków (`pthread_join`):

- pthread_timedjoin_np (3) — oczekiwanie ograniczone czasowo,
- pthread_tryjoin_np (3) — sprawdzenie, bez oczekiwania, czy wątek się zakończył

### 5.1.5 Zrzeczenie się czasu procesora

- `pthread_yield (3)` — alias dla standardowego sched_yield, czyli zrzeczenia się czasu procesora i oczekiwanie na ponowne jego przyznanie

## 5.2 Cygwin

Cygwin definiuje dwie dodatkowe funkcje:

1. `pthread_suspend(pthread_t id)` — wstrzymanie wykonywania wskazanego wątku;
2. `pthread_resume(pthread_t id)` — wznowienie wykonywania uprzednio wstrzymanego wątku.

# Rozdział 6

# Przykładowe programy

## 6.1 Kompilacja

Wszystkie programy zostały skompilowane kompilatorem `gcc` i uruchomione na Linuxie 2.6, część bez problemów skompilowała się i uruchomiła w Cygwinie.

Kompilator był wywoływany z następującymi opcjami:

```
gcc -Wall -pedantic -std=c99 \textbf{-lpthreads} \emph{program.c} -o przyklad
```

lub

```
gcc -Wall -pedantic -std=c99 \textbf{-pthread} \emph{program.c} -o przyklad
```

Niektóre przykłady (wirujące blokady, bariery) muszą być linkowane z `librt`, czyli konieczna jest opcja `-lrt`.

Aby skompilować część z nich trzeba ustawić odpowiednie definicje preprocesora, w szczególności `_POSIX_C_SOURCE` na odpowiednią wartość — w przykładach jest to `200809L`, dając dostęp do funkcji z nowszych rewizji standardu POSIX.

## 6.2 Wykaz przykładów

**Program 1** (Tworzenie wątku)

- `pthread_create`
- `pthread_join`

**Program 2** (Przekazywanie argumentów)

- `pthread_create`
- `pthread_join`

**Program 3** (Kończenie wątku)

- `pthread_exit`
- `pthread_create`
- `pthread_join`

**Program 4** (Atrybuty wątku)

- `pthread_attr_getdetachstate`
- `pthread_attr_getguardsize`
- `pthread_attr_getinheritsched`
- `pthread_attr_getschedparam`
- `pthread_attr_getschedpolicy`
- `pthread_attr_getscope`
- `pthread_attr_getstackaddr`
- `pthread_attr_getstacksize`
- `pthread_attr_init`

**Program 5** (Stos funkcji finalizujących (cleanup))

- `pthread_cleanup_pop`
- `pthread_cleanup_push`
- `pthread_create`

- `pthread_exit`
- `pthread_join`

**Program 6** (Funkcje wywoływane jednokrotnie)

- `pthread_once`
- `pthread_create`
- `pthread_join`

**Program 7** (Przerywanie wątków)

- `pthread_cancel`
- `pthread_setcancelstate`
- `pthread_setcanceltype`
- `pthread_testcancel`
- `pthread_cleanup_push`
- `pthread_create`
- `pthread_join`

**Program 8** (Typy mutexów)

- `pthread_mutexattr_settype`
- `pthread_mutexattr_init`
- `pthread_mutex_unlock`
- `pthread_mutex_lock`
- `pthread_create`
- `pthread_join`

**Program 9** (Zmienne warunkowe)

- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_create`

**Program 10** (Lokalne dane wątku)

- `pthread_key_create`
- `pthread_key_delete`
- `pthread_setspecific`
- `pthread_getspecific`
- `pthread_create`
- `pthread_join`

**Program 11** (Bariery)

- `pthread_barrier_init`
- `pthread_barrier_destroy`

- `pthread_barrier_wait`
- `pthread_create`

**Program 12** (Rozmiar i adres stosu)

- `pthread_attr_setstacksize`
- `pthread_attr_getstacksize`
- `pthread_attr_init`
- `pthread_attr_destroy`
- `pthread_create`
- `pthread_join`

**Program 13** (UNIX-owe sygnały)

- `pthread_kill`
- `pthread_sigmask`
- `pthread_create`
- `pthread_self`

**Program 14** (Czas procesora zużyty przez wątek)

- `pthread_getcpuclockid`
- `clock_gettime` (nie pthreads)
- `pthread_create`
- `pthread_join`

**Program 15** (Pozyskiwanie i zwolnienie blokady)

- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutex_init`
- `pthread_create`
- `pthread_join`

**Program 16** (Blokady zapis/odczyt)

- `pthread_rwlock_init`
- `pthread_rwlock_rdlock`
- `pthread_rwlock_wrlock`
- `pthread_rwlock_unlock`
- `pthread_create`

**Program 17** (Synchronizacja między wątkami różnych procesów)

- `pthread_cond_init`
- `pthread_cond_signal`
- `pthread_cond_wait`
- `pthread_condattr_init`
- `pthread_condattr_setpshared`
- `pthread_mutex_init`

- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutexattr_init`
- `pthread_mutexattr_setpshared`

**Program 18** (Pthreads i forkowanie)

- `pthread_atfork`
- `pthread_create`

**Program 19** (Wybór algorytmu szeregowania i ustawienie jego parametrów)

- `pthread_attr_setinheritsched`
- `pthread_attr_setschedpolicy`
- `pthread_attr_setschedparam`
- `sched_get_priority_min`
- `sched_get_priority_max`
- `pthread_attr_init`
- `pthread_attr_destroy`
- `pthread_create`
- `pthread_join`

## Przykład 1

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: hello world z kilku wątków
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

/* funkcja wykonywana w wątku - nic specjalnego nie robi */
void* watek(void* _arg) {
  puts("Witaj w równoległym świecie!");
  return NULL;
}
//----------------------------------------------------------------------

#define N 5 /* liczba wątków */

int main() {
  pthread_t id[N];
  int i;

  /* utworzenie kilku wątków wątku */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id[i], NULL, watek, NULL);
    test_errno("Nie powiodło się pthread_create");
  }

  /* oczekiwanie na jego zakończenie */
  for (i=0; i < N; i++) {
    errno = pthread_join(id[i], NULL);
    test_errno("pthread_join");
  }

  return EXIT_SUCCESS;
}
//----------------------------------------------------------------------
```

## Przykład 3

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat:  zakończenie wątku z poziomu funkcji wywływanych w wątku
 *    za pomocą
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)


void koniec_watku(int licznik, int limit) {
  int i;
  for (i=0; i < licznik; i++) putchar(' ');
  printf("licznik = %d, limit = %d\n", licznik, limit);

  if (licznik == limit)
    /* zakończenie wątku w którego kontekście wykonywana jest ta funkcja */
    pthread_exit(NULL);
  else
    koniec_watku(licznik+1, limit);
}
//----------------------------------------------------------------------

void* watek(void* arg) {
  koniec_watku(0, 5);
  return NULL;
}
//----------------------------------------------------------------------

int main() {
  pthread_t id;

  /* utworzenie wątku */
  errno = pthread_create(&id, NULL, watek, NULL);
  test_errno("pthread_create");

  /* oczekiwanie na jego zakończenie */
  errno = pthread_join(id, NULL);
  test_errno("pthread_join");

  return EXIT_SUCCESS;
}
```

//————————————————————————————————————————————————————————————————————————————————

## Przykład 2

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: przekazywanie parametrów do funkcji wątku i zwracanie wyników
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

typedef struct _Arg { // struktura argumentów dla wątku 1.
  char  napis[256];
  int rok;
  int mies;
  int dzien;
} Argument;

void* watek1(void* _arg) {
  Argument* arg = (Argument*)_arg;

  printf("Witaj %s w dniu %04d-%02d-%02d\n",
    arg->napis,
    arg->rok,
    arg->mies,
    arg->dzien
  );

  return NULL;
}
//---------------------------------------------------------------------------

/* wątek 2. zwraca pewien dynamicznie tworzony napis */
void* watek2(void* liczba) {
  char* napis;
  int i;
  printf("Wątek 2 wywołany z argumentem liczbowym %d\n", (int)liczba);

  napis = malloc((int)liczba + 1);
  if (napis) {
    for (i=0; i < (int)liczba; i++)
      napis[i] = 'x';

    napis[(int)liczba] = 0;
```

```
  }
  return napis;
}
//---------------------------------------------------------------------------

int main() {
  pthread_t w1, w2;
  Argument arg;
  char* wynik;

  /* przygotowanie argumentów */
  strcpy(arg.napis, "Wikibooks");
  arg.rok  = 2010;
  arg.mies = 3;
  arg.dzien = 14;

  /* utworzenie dwóch wątków */
  errno = pthread_create(&w1, NULL, watek1, &arg);
  test_errno("pthread_create");

  errno = pthread_create(&w2, NULL, watek2, (void*)27);
  test_errno("pthread_create");

  /* oczekiwanie na zakończenie obu */
  errno = pthread_join(w1, NULL);
  test_errno("pthread_join");

  errno = pthread_join(w2, (void**)&wynik);
  test_errno("pthread_join");

  if (wynik) {
    printf("wątek 2 zwrócił napis: '%s'\n", wynik);
    free(wynik);
  }
  else
    puts("wątek 2 nic nie zwrócił");

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 4

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: atrybuty wątku - wypisanie domyślnych wartości
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <limits.h> // PTHREAD_STACK_MIN
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

void wyswietl_atrybuty(const pthread_attr_t* attr) {
  int x;
  size_t rozmiar;
  void* addr;
  struct sched_param param;

  puts("atrybuty wątku");

  // rodzaj wątku
  printf("* rodzaj: ");
  errno = pthread_attr_getdetachstate(attr, &x);
  test_errno("pthread_attr_getdetachstate");

  switch (x) {
    case PTHREAD_CREATE_JOINABLE:
      puts("joinable");
      break;
    case PTHREAD_CREATE_DETACHED:
      puts("detached");
      break;
    default:
      puts("???");
  }

  // adres i rozmiar stosu
  errno = pthread_attr_getstackaddr(attr, &addr);
  test_errno("pthread_attr_getstackaddr");
  printf("* adres stosu: %p\n", addr);

  errno = pthread_attr_getstacksize(attr, &rozmiar);
  test_errno("pthread_attr_getstacksize");
```

```
printf("* rozmiar stosu: %d (minimalny %d)\n", rozmiar, PTHREAD_STACK_MIN
    );

// rozmiar obszaru zabezpieczającego stosu
errno = pthread_attr_getguardsize(attr, &rozmiar);
test_errno("pthread_attr_getguardsize");
printf("* rozmiar obszaru zabezpieczającego: %d\n", rozmiar);

// szeregowanie
errno = pthread_attr_getinheritsched(attr, &x);
test_errno("pthread_attr_getinheritsched");
switch (x) {
  case PTHREAD_INHERIT_SCHED:
    puts("* parametry szeregowania dziedziczone");
    break;

  case PTHREAD_EXPLICIT_SCHED:
    puts("* parametry podawane bezpośrednio");

    //
    printf("   - algorytm szeregowania: ");
    errno = pthread_attr_getschedpolicy(attr, &x);
    test_errno("pthread_attr_getschedpolicy");
    switch (x) {
      case SCHED_OTHER:
        puts("SCHED_OTHER");
        break;
      case SCHED_RR:
        puts("SCHED_RR");
        break;
      case SCHED_FIFO:
        puts("SCHED_FIFO");
        break;
      default:
        puts("???");
    }

    //
    errno = pthread_attr_getschedparam(attr, &param);
    test_errno("pthread_attr_getschedparam");
    printf("   - priorytet: %d\n", param.sched_priority);
    break;

  default:
    puts("???");
}

// zakres szeregowania
errno = pthread_attr_getscope(attr, &x);
test_errno("pthread_attr_getscope");

printf("* zakres szeregowania: ");
switch (x) {
  case PTHREAD_SCOPE_PROCESS:
```

```
      puts("proces");
      break;
    case PTHREAD_SCOPE_SYSTEM:
      puts("system");
      break;
    default:
      puts("???");
  }
}
//---------------------------------------------------------------------------

int main() {
  pthread_attr_t attr;

  errno = pthread_attr_init(&attr);
  test_errno("pthread_attr_init");

  wyswietl_atrybuty(&attr);
  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 5

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: funkcje finalizujące (cleanup)
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h> // sleep

#define test_errno(info) do {if (errno) {perror(info); exit(EXIT_FAILURE)
    ;}} while(0)

/* funkcja finalizująca */
void zwolnij_pamiec(void* adres) {
  printf("zwalnianie pamięci spod adresu %p\n", adres);
  free(adres);
}
//------------------------------------------------------------------------

void* watek(void* id) {
  char* tablica1 = malloc(100);
  char* tablica2 = NULL;
  printf("wątek #%d zaalokował 100 bajtów pod adresem %p\n", (int)id,
      tablica1);

  pthread_cleanup_push(zwolnij_pamiec, tablica1);
  if (tablica1) {
    tablica2 = malloc(200);
    printf("wątek #%d zaalokował 200 bajtów pod adresem %p\n", (int)id,
        tablica2);
    pthread_cleanup_push(zwolnij_pamiec, tablica2);

    if ((int)id > 0)
      /* wątek się kończy w tym punkcie, funkcje finalzujące
         zostaną uruchomione */
      pthread_exit(NULL);

    pthread_cleanup_pop(1);
  }

  pthread_cleanup_pop(1);

  printf("wątek #%d zakończył się\n", (int)id);
  return NULL;
}
```

```
//------------------------------------------------------------------------------

int main() {
  pthread_t id1;
  pthread_t id2;

  /* utworzenie 2 wątków */
  errno = pthread_create(&id1, NULL, watek, (void*)(0));
  test_errno("pthread_create (1)");

  errno = pthread_create(&id2, NULL, watek, (void*)(1));
  test_errno("pthread_create (2)");

  /* oczekiwanie na zakończenie */
  pthread_join(id1, NULL);
  pthread_join(id2, NULL);

  return EXIT_SUCCESS;
}
//------------------------------------------------------------------------------
```

## Przykład 6

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: funkcje wykonywane jednokrotnie
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

/* obiekt gwarantujący jednokrotne wykonanie, musi zostać zainicjowany */
pthread_once_t program_gotowy = PTHREAD_ONCE_INIT;

void inicjalizacja() {
  /* inicjalizacja, np. prekalkulowanie jakiś tablic,
     otwieranie pliku logowania itp. */
  puts("Rozpoczynanie programu");
}
//----------------------------------------------------------------------

void* watek(void* numer) {
  pthread_once(&program_gotowy, inicjalizacja);

  printf("Uruchomiono watek nr %d\n", (int)numer);
  return NULL;
}
//----------------------------------------------------------------------

#define N 10 /* liczba wątków */

int main() {
  pthread_t id[N];
  int i;

  /* utworzenie wątków */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id[i], NULL, watek, (void*)i);
    test_errno("pthread_create");
  }

  /* oczekiwanie na jego zakończenie */
  for (i=0; i < N; i++) {
    errno = pthread_join(id[i], NULL);
    test_errno("pthread_join");
```

```
    }

    return EXIT_SUCCESS;
}
//-------------------------------------------------------------------------------
```

## Przykład 7

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat:  przerywanie wątków - program tworzy 3 wątki z różnymi
 *    ustawieniami dotyczącymi przerywania:
 *    1) dopuszcza przerwanie asynchroniczne
 *    2) dopuszcza przerwanie opóźnione
 *    3) przez pewien czas w ogóle blokuje przerwania
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h> // pause w watek3
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

void zakonczenie(void* numer) {
  printf("funkcja finalizująca dla wątku #%d\n", (int)numer);
}
//----------------------------------------------------------------------

void* watek1(void* numer) {
  int i, n;

  pthread_cleanup_push(zakonczenie, numer);

  errno = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  test_errno("pthread_setcancelstate");

  errno = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  test_errno("pthread_setcanceltype");

  printf("\turuchomiono wątek #%d (przerwanie asynchroniczne)\n", (int)
      numer);
  while (1) {
    n = 1000000;
    for (i=0; i < n; i++)
      /**/;
  }

  pthread_cleanup_pop(1);
  return NULL;
}
//----------------------------------------------------------------------
```

```
void* watek2(void* numer) {
  int i, n;
  pthread_cleanup_push(zakonczenie, numer);

  errno = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  test_errno("pthread_setcancelstate");

  errno = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
  test_errno("pthread_setcanceltype");

  printf("\turuchomiono wątek #%d (przerwanie opóźnione)\n", (int)numer);
  while (1) {
    pthread_testcancel(); // punkt przerwania
    n = 1000000;
    for (i=0; i < n; i++)
      /**/;
  }

  pthread_cleanup_pop(1);
  return NULL;
}
//---------------------------------------------------------------------------

void* watek3(void* numer) {
  pthread_cleanup_push(zakonczenie, numer);
  errno = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  test_errno("pthread_setcancelstate");

  printf("\turuchomiono wątek #%d (przez 2 sekundy nie można przerwać)\n",
      (int)numer);

  sleep(2);

  printf("\twątek #%d można już przerwać\n", (int)numer);
  errno = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  test_errno("pthread_setcancelstate");
  pause();

  pthread_cleanup_pop(1);
  return NULL;
}
//---------------------------------------------------------------------------

void przerwanie(pthread_t id, const char* napis) {
  printf("%s: wysyłanie sygnału przerwania do wątku\n", napis);
  errno = pthread_cancel(id);
  test_errno("pthread_cancel");

  printf("%s: wysłano, oczekiwanie na zakończenie\n", napis);
  errno = pthread_join(id, NULL);
  test_errno("pthread_join");

  printf("%s: wątek zakończony\n", napis);
```

```
}
//---------------------------------------------------------------------------

int main() {
  pthread_t id[3];

  /* utworzenie wątków */
  errno = pthread_create(&id[0], NULL, watek1, (void*)(0));
  test_errno("pthread_create (1)");

  errno = pthread_create(&id[1], NULL, watek2, (void*)(1));
  test_errno("pthread_create (2)");

  errno = pthread_create(&id[2], NULL, watek3, (void*)(2));
  test_errno("pthread_create (3)");

  /* przerywanie kolejnych wątków */
  przerwanie(id[0], "#0");
  przerwanie(id[1], "#1");
  przerwanie(id[2], "#2");

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 8

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat:  różne zachowanie mutexów w pthreads przy próbie ponownego
 *    założenia blokady
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define __USE_UNIX98
#include <pthread.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

pthread_t id;
pthread_mutex_t mutex;
pthread_mutexattr_t mutexattr;

void* watek(void* _arg) {
  int errno;

  // 1
  puts("przed wykonaniem pthread_mutex_lock (1)");
    errno = pthread_mutex_lock(&mutex);
    test_errno("pthread_mutex_lock (1)");
  puts("... wykonano pthread_mutex_lock (1)");

  // 2
  puts("przed wykonaniem pthread_mutex_lock (2)");
    errno = pthread_mutex_lock(&mutex);
    test_errno("pthread_mutex_lock (2)");
  puts("... wykonano pthread_mutex_lock (2)");

  // 3
  puts("przed wykonaniem pthread_mutex_unlock (2)");
    errno = pthread_mutex_unlock(&mutex);
    test_errno("pthread_mutex_unlock (2)");
  puts("... wykonano pthread_mutex_unlock (2)");

  // 4
  puts("przed wykonaniem pthread_mutex_unlock (1)");
    errno = pthread_mutex_unlock(&mutex);
    test_errno("pthread_mutex_unlock (1)");
  puts("... wykonano pthread_mutex_unlock (1)");
```

```
  return NULL;
}
//---------------------------------------------------------------------------

int main(int argc, char* argv[]) {
  int errno;

  pthread_mutexattr_init(&mutexattr);

  if (argc > 1) {
    switch (atoi(argv[1])) {
      case 1:
        puts("mutex typu PTHREAD_MUTEX_ERRORCHECK");
        pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_ERRORCHECK);
        break;
      case 2:
        puts("mutex typu PTHREAD_MUTEX_RECURSIVE");
        pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_RECURSIVE);
        break;
      default:
        puts("mutex typu PTHREAD_MUTEX_NORMAL");
        pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_NORMAL);
        break;
    }
  }
  else {
    puts("użycie: program [0|1|2]");
    return EXIT_FAILURE;
  }

  /* inicjalizacja mutexu */
  errno = pthread_mutex_init(&mutex, &mutexattr);
  test_errno("pthread_mutex_init");

  /* utworzenie wštku */
  errno = pthread_create(&id, NULL, watek, NULL);
  test_errno("pthread_create");

  /* oczekiwanie na jego zakończenie */
  pthread_join(id, NULL);
  test_errno("pthread_join");

  puts("program zakończony");
  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 9

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: zmienne warunkowe
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h> // sleep

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;

char warunek = 0;

void* watek(void* numer) {
  printf("\turuchomiono wątek #%d\n", (int)numer);
  while (1) {
    pthread_mutex_lock(&mutex);
    do {
      if (warunek)
        break;
      else {
        printf("\twątek #%d oczekuje na sygnał...\n", (int)numer);
        pthread_cond_wait(&cond, &mutex);
        printf("\t... wątek #%d otrzymał sygnał!\n", (int)numer);
      }
    } while (1);
    pthread_mutex_unlock(&mutex);
    /* ... */
  }

  return NULL;
}

#define N 5 /* liczba wątków */

int main() {
  pthread_t id[N];
  int i;

  puts("początek programu");

  /* utworzenie wątków */
  for (i=0; i < N; i++) {
```

```
    errno = pthread_create(&id[i], NULL, watek, (void*)(i+1));
    if (errno) {
      perror("pthread_create");
      return EXIT_FAILURE;
    }
  }

  /* wysyłanie sygnałów */

  sleep(1);
  puts("pthread_cond_signal - sygnalizacja");
  pthread_cond_signal(&cond);

  sleep(1);
  puts("pthread_cond_broadcast - rozgłaszanie");
  pthread_cond_broadcast(&cond);

  sleep(1);

  /* kończymy proces, bez oglądania się na wątki */
  puts("koniec programu");
  return EXIT_SUCCESS;
}
```

## Przykład 10

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: prywatne dane wątków
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

#define test_errno(info) do {if (errno) {perror(info); exit(EXIT_FAILURE)
    ;}} while(0)

pthread_key_t klucz;

/* funkcja wypsuje wiersz, poprzedzając go prefiksem przypisanym do wątku
   */
void wyswietl(const char* napis) {
  char* prefiks = (char*)pthread_getspecific(klucz);
  if (prefiks == NULL)
    /* należy zabezpieczyć się przed sytuacją, gdy wywołujący
       wątek nie przyporządkował nic do klucza */
    puts(napis);
  else
    printf("%s: %s\n", prefiks, napis);
}
//
   ------------------------------------------------------------------------


/* destruktor klucza */
void destruktor(void* napis) {
  printf("wywołano destruktor, adres pamięci do zwolnienia: %p ('%s')\n",
    napis,
    (char*)napis
  );
  free(napis);
}
//
   ------------------------------------------------------------------------


void* watek(void* napis) {
  /* ustawienie prefiksu w lokalnych danych wątku */
  int status = pthread_setspecific(klucz, napis);
```

```
  if (status)
    fprintf(stderr, "pthread_setspecific: %s\n", strerror(status));
  else
    printf("adres napisu: %p ('%s')\n", napis, (char*)napis);

  wyswietl("Witaj w równoległym świecie!");
  sleep(1);
  wyswietl("Wątek wykonuje pracę");
  sleep(1);
  wyswietl("Wątek zakończony");
  return NULL;
}
//
```

---

```
char* strdup(const char*);

#define N 3

int main() {
  pthread_t id[N];
  int i;
  char* prefiks[3] = {"***", "!!!", "###"}; // prefiksy dla komunikatów z
      wątków

  /* utworzenie klucza */
  errno = pthread_key_create(&klucz, destruktor);
  test_errno("pthread_key_create");

  /* utworzenie wątków */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id[i], NULL, watek, (void*)strdup(prefiks[i %
        3]));
    test_errno("pthread_create");
  }

  /* oczekiwanie na ich zakończenie */
  for (i=0; i < N; i++)
    pthread_join(id[i], NULL);

  /* usunięcie klucza */
  errno = pthread_key_delete(klucz);
  test_errno("pthread_key_delete");

  return EXIT_SUCCESS;
}
//
```

---

```
char* strdup(const char* s) {
  char *d = NULL;
  if (s) {
```

```
    d = (char*)malloc(strlen(s)+1);
    if (d)
      strcpy(d, s);
  }
  return d;
}
//
```

## Przykład 11

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: bariery
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

pthread_barrier_t bariera;

void* watek(void* numer) {
  int s, status;

  s = rand() % 4 + 1; // oczekiwanie 1-4 s
  printf("\twątek #%d rozpoczęty, zostanie wstrzymany na %d sekund\n", (int
      )numer, s);

  sleep(s);

  printf("\twątek #%d osiągnął barierę\n", (int)numer);
  status = pthread_barrier_wait(&bariera);
  switch (status) {
    case 0: // ok
      break;

    case PTHREAD_BARRIER_SERIAL_THREAD:
      printf(
        "\twszystkie wątki osiągnęły barierę "
        "(PTHREAD_BARRIER_SERIAL_THREAD w wątku #%d)\n",
        (int)numer
      );
      break;

    default:
      fprintf(stderr, "pthread_barrier_wait: %s\n", strerror(status));
      break;
  }
  return NULL;
```

```
}
//---------------------------------------------------------------------------

#define N 10  /* liczba wątków */

int main() {
  int i;
  pthread_t id[N];

  srand(time(NULL));

  printf("zostanie uruchomionych %d wątków\n", N);

  /* inicjalizacja bariery - N wątków */
  errno = pthread_barrier_init(&bariera, NULL, N);
  test_errno("pthread_barrier_init");

  /* utworzenie N wątków */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id[i], NULL, watek, (void*)i);
    test_errno("pthread_create");
  }

  /* oczekiwaie na dojście do bariery wszystkich wątków */
  for (i=0; i < N; i++) {
    errno = pthread_join(id[i], NULL);
    test_errno("pthread_join");
  }

  /* zwolnienie bariery */
  errno = pthread_barrier_destroy(&bariera);
  test_errno("pthread_barrier_destroy");

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 12

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: zmiana rozmiaru stosu wątku
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

#define N (100*1024)

/* wątek używa sporej tablicy alokowanej na stosie */
void* watek(void* arg) {
  char tablica[N];
  int i;
  for (i=0; i < N; i++)
    tablica[i] = 0;

  return NULL;
}
//----------------------------------------------------------------------

int main(int argc, char* argv[]) {
  pthread_t id;
  pthread_attr_t  attr;
  size_t     rozmiar;

  errno = pthread_attr_init(&attr);
  if (errno) {
    perror("pthread_attr_init");
    return EXIT_FAILURE;
  }

  if (argc > 1) {
    rozmiar = atoi(argv[1]);
    printf("rozmiar stosu ustalony przez użytkownika: %u\n", rozmiar);
    printf("minimalny rozmiar stosu: %u\n", PTHREAD_STACK_MIN);

    errno = pthread_attr_setstacksize(&attr, rozmiar);
    test_errno("pthread_attr_setstacksize");
```

```
  }
  else {
    pthread_attr_getstacksize(&attr, &rozmiar);
    printf("domyślny rozmiar stosu: %u\n", rozmiar);
  }

  errno = pthread_create(&id, &attr, watek, NULL);
  test_errno("pthread_create");

  pthread_join(id, NULL);
  puts("wątek zakończony");

  pthread_attr_destroy(&attr);
}
//---------------------------------------------------------------------------
```

## Przykład 13

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: wysyłanie sygnałow UNIX-owych do wątków
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <unistd.h> // sleep

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)


pthread_t main_id;  // id głównego wątek

// funkcja wątku
void* watek(void* nieuzywany) {
  puts("\twątek się rozpoczął");
  sleep(1);

  puts("\twątek wysyła sygnał SIGUSR1 do głównego wątku");
  errno = pthread_kill(main_id, SIGUSR1);
  test_errno("pthread_kill");

  return NULL;
}
//------------------------------------------------------------------------------

int main() {
  pthread_t id;
  int signum;
  sigset_t mask;

  // blokowanie SIGUSR1
  sigemptyset(&mask);
  sigaddset(&mask, SIGUSR1);

  errno = pthread_sigmask(SIG_BLOCK, &mask, NULL);
  test_errno("pthread_kill");

  // odczyt id głównego wątku
  main_id = pthread_self();
```

```
  // utworzenie wątku
  errno = pthread_create(&id, NULL, watek, NULL);
  test_errno("pthread_create");


  // oczekiwanie na sygnał
  puts("wątek główny oczekuje na sygnał");

  sigwait(&mask, &signum);
  test_errno("sigwait");
  if (signum == SIGUSR1)
    puts("wątek główny otrzymał sygnał SIGUSR1");
  else
    printf("wątek główny otrzymał sygnał %d\n", signum);

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 14

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: odczyt czasu CPU, jaki zużył wątek
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#include <pthread.h>
#include <unistd.h>
#include <time.h> // sleep
#include <string.h> // strerror

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

// funkcja zwraca czas w milisekundach dla wskazanego zegara
long clock_ms(const clockid_t id_zegara);

// funkcja zwraca czas CPU dla wątku (w milisekundach)
long get_thread_time(pthread_t id);

/* parametry wątku */
typedef struct {
  int id; // numer
  int n;  // liczba iteracji
} parametry;

// funkcja wątku
void* watek(void* _arg) {
  parametry* arg = (parametry*)_arg;
  int i;
  printf("wątek #%d uruchomiony, dwa razy wykona %d pustych pętli\n",
    (int)arg->id,
    (int)arg->n
  );

  for (i=0; i < arg->n; i++)
    /* zużycie czasu procesora */;

  sleep(2);

  for (i=0; i < arg->n; i++)
    /* zużycie czasu procesora */;
```

```
  /* podsumowanie pracy */
  printf("wątek #%d zakończony, zużył %ldms czasu procesora\n",
    (int)arg->id,
    clock_ms(CLOCK_THREAD_CPUTIME_ID)
  );
  return NULL;
}
//---------------------------------------------------------------------------

#define N 10  // liczba wątków

int main() {
  pthread_t id[N];
  parametry param[N];
  int i;

  srand(time(NULL));

  printf("początek programu, uruchomianie zostanie %d wątków\n", N);

  /* utworzenie wątku */
  for (i=0; i < N; i++) {
    param[i].id = i;
    param[i].n = rand() % 100000000 + 1;

    errno = pthread_create(&id[i], NULL, watek, &param[i]);
    test_errno("pthread_create");
  }

  /* stan na mniej więcej półmetku */
  sleep(1);
  puts("po około sekundzie wątki zużyły:");
  for (i=0; i < N; i++)
    printf("* #%d: %ldms\n", i, get_thread_time(id[i]));

  /* oczekiwanie na zakończenie wątków */
  for (i=0; i < N; i++) {
    errno = pthread_join(id[i], NULL);
    test_errno("pthread_join");
  }

  /* jeszcze podsumowanie */
  puts("");
  printf("główny wątek zużył %ldms czasu procesora\n", clock_ms(
      CLOCK_THREAD_CPUTIME_ID));
  printf("proces zużył %ldms czasu procesora\n", clock_ms(
      CLOCK_PROCESS_CPUTIME_ID));

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------

long get_thread_time(pthread_t id) {
```

```
  clockid_t id_zegara;

  errno = pthread_getcpuclockid(id, &id_zegara);
  test_errno("pthread_getcpuclockid");

  return clock_ms(id_zegara);
}
//------------------------------------------------------------------------

long clock_ms(const clockid_t id_zegara) {
  struct timespec czas;
  if (clock_gettime(id_zegara, &czas)) {
    perror("clock_gettime");
    exit(EXIT_FAILURE);
  }
  else
    return (czas.tv_sec * 1000) +
           (czas.tv_nsec/1000000);
}
//------------------------------------------------------------------------
```

## Przykład 15

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat:  sekcja krytyczna z użyciem mutexów
 *    jeśli zdefiniowane zostanie BLOKADA, mutex blokuje
 *    dostęp do zmiennej, w przeciwnym razie wątki zmieniają
 *    ją bez żadnej synchronizacji, co może prowadzić do błędu
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)


#define N 10  /* liczba wątków */
#define K 1000  /* liczba iteracji (z tą wartością należy eksperymentować)
    */

pthread_mutex_t blokada;
int licznik = 0;     // globalny licznik, powinien być chroniony blokadą

void ms_sleep(unsigned ms) {
  struct timespec req;
  req.tv_sec  = (ms / 1000);
  req.tv_nsec = (ms % 1000 * 1000000);
  nanosleep(&req, NULL);
}
//---------------------------------------------------------------------------

void* watek(void* numer) {
  int i;
  for (i=0; i < K; i++) {
#ifdef BLOKADA
    errno = pthread_mutex_lock(&blokada);
    test_errno("pthread_mutex_lock");
#endif
    licznik = licznik + 1;
    ms_sleep(1);
#ifdef BLOKADA
    errno = pthread_mutex_unlock(&blokada);
    test_errno("pthread_mutex_unlock");
#endif
  }
```

```
  return NULL;
}
//---------------------------------------------------------------------------

int main() {
  pthread_t id[N];
  int i;

  printf("licznik = %d\n", licznik);

  errno = pthread_mutex_init(&blokada, NULL);
  test_errno("pthread_mutex_init");

  /* utworzenie wątku */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id[i], NULL, watek, (void*)i);
    test_errno("pthread_create");
  }

  /* oczekiwanie na jego zakończenie */
  for (i=0; i < N; i++) {
    errno = pthread_join(id[i], NULL);
    test_errno("pthread_join");
  }

  printf("licznik = %d, spodziewana wartość = %d %s\n",
    licznik,
    N*K,
    (licznik != N*K ? "BŁĄD!!!" : "")
  );

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 16

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: blokady zapis/odczy (rwlock)
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <time.h>

void ms_sleep(const unsigned ms);
#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)


pthread_rwlock_t  blokada;
int      wartosc;  // obiekt chroniony blokadą

/* wątek zmienia wartość */
void* pisarz(void* numer) {
  while (1) {
    printf("␣pisarz␣#%d␣czeka␣na␣dostęp\n", (int)numer);
    errno = pthread_rwlock_wrlock(&blokada);
    test_errno("pthread_rwlock_wrlock");
    printf("␣pisarz␣#%d␣ustawia␣nową␣wartość\n", (int)numer);

    ms_sleep(113);

    printf("␣pisarz␣#%d␣zwalnia␣blokadę\n", (int)numer);
    errno = pthread_rwlock_unlock(&blokada);
    test_errno("pthread_rwlock_unlock");

    ms_sleep(317);

  }

  return NULL;
}
//----------------------------------------------------------------------

/* wątek tylko odczytuje wartość */
void* czytelnik(void* numer) {
  int errno;
  while (1) {
    printf("␣␣czytelnik␣#%d␣czeka␣na␣dostęp\n", (int)numer);
```

```
    errno = pthread_rwlock_rdlock(&blokada);
    test_errno("pthread_rwlock_rdlock");
    printf("  czytelnik #%d odczytuje wartość\n", (int)numer);

    ms_sleep(13);

    printf("  czytelnik #%d zwalnia blokadę\n", (int)numer);
    errno = pthread_rwlock_unlock(&blokada);
    test_errno("pthread_rwlock_unlock");

    ms_sleep(13);

  }
  return NULL;
}
//----------------------------------------------------------------------

#define N 5 /* liczba wątków */
#define K 2

int main() {
  pthread_t id;
  int i;

  pthread_rwlock_init(&blokada, NULL);

  /* utworzenie K wątków piszących */
  for (i=0; i < K; i++) {
    errno = pthread_create(&id, NULL, pisarz, (void*)i);
    test_errno("pthread_create");
  }

  /* utworzenie N wątków czytających */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id, NULL, czytelnik, (void*)i);
    test_errno("pthread_create");
  }

  /* kilka sekund na pracę wątków i koniec */
  ms_sleep(1500);

  return EXIT_SUCCESS;
}
//----------------------------------------------------------------------

void ms_sleep(const unsigned ms) {
  struct timespec req;
  req.tv_sec  = (ms / 1000);
  req.tv_nsec = (ms % 1000 * 1000000);
  nanosleep(&req, NULL);
}
//----------------------------------------------------------------------
```

## Przykład 17

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat:  synchronizacja między wątkami różnych procesów
 *    współdzielony mutex i zmienna warunkowa
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _XOPEN_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <sys/shm.h>

#include <errno.h>
#include <unistd.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

struct pamiec_dzielona {
  /* współdzielone mutex i zmienna warunkowa */
  pthread_mutex_t mutex;
  pthread_cond_t  cond;
  pthread_mutexattr_t mattr;
  pthread_condattr_t  cattr;

  /* dane */
  char napis_dostepny;
  char napis[256];
} *pamiec;


/* proces 1 tworzy segment pamięci wspólnej, tworzy mutex i zmienną
    warunkową WSPÓŁDZIELONE
    z innymi procesami, po czym oczekuje na sygnał, aż napis stanie się
       dostępny */
void proces1() {
  int shmid;
  int errno;

  shmid = shmget(IPC_PRIVATE, sizeof(pamiec), 0666);
  test_errno("shm_open");
  printf("id segmentu pamięci dzielonej: %d\n", shmid);

  pamiec = shmat(shmid, 0, 0);
  test_errno("shmat");
  printf("adres przyłączonego segmentu: %p\n", (void*)pamiec);
```

```
  /* inicjalizacja */
  pamiec->napis_dostepny  = 0;
  memset(pamiec->napis, 0, sizeof(pamiec->napis));

  /* tworzenie mutexu i zmiennej warunkowej */
  errno = pthread_mutexattr_init(&pamiec->mattr);
  test_errno("pthread_mutexattr_init");

  errno = pthread_mutexattr_setpshared(&pamiec->mattr,
     PTHREAD_PROCESS_SHARED);
  test_errno("pthread_mutexattr_setpshared");

  errno = pthread_mutex_init(&pamiec->mutex, &pamiec->mattr);
  test_errno("pthread_mutex_init");

  /* tworzenie mutexu i zmiennej warunkowej */
  errno = pthread_condattr_init(&pamiec->cattr);
  test_errno("pthread_condattr_init");

  errno = pthread_condattr_setpshared(&pamiec->cattr,
     PTHREAD_PROCESS_SHARED);
  test_errno("pthread_condattr_setpshared");

  errno = pthread_cond_init(&pamiec->cond, &pamiec->cattr);
  test_errno("pthread_cond_init");

  /* oczekiwanie na ustawienie napisu przez inny proces */
  errno = pthread_mutex_lock(&pamiec->mutex);
  test_errno("pthread_mutex_lock");
  do {
    if (pamiec->napis_dostepny) {
      printf("inny proces ustawił napis: '%s'\n", pamiec->napis);
      break;
    }
    else {
      puts("pthread_cond_wait");
      errno = pthread_cond_wait(&pamiec->cond, &pamiec->mutex);
      test_errno("pthread_cond_wait");
    }
  } while (1);

  errno = pthread_mutex_unlock(&pamiec->mutex);
  test_errno("pthread_mutex_unlock");

  /* odłączenie od segmentu pamięci */
  shmdt(pamiec);
  test_errno("shmdt");

  /* i skasowanie go */
  shmctl(shmid, IPC_RMID, NULL);

  return;
}
```

```
//---------------------------------------------------------------------------

/* proces 2 przyłącza się do segmentu i używając współdzielonego mutexu i
    zmiennej
    warunkowej ustawia napis i sygnalizuje go procesowi 1 funkcją
       pthread_cond_signal */
void proces2(int shmid, const char* napis) {
  int errno;

  pamiec = shmat(shmid, 0, 0);
  test_errno("shmat");
  printf("adres przyłączonego segmentu: %p\n", (void*)pamiec);

  errno = pthread_mutex_lock(&pamiec->mutex);
  test_errno("pthread_mutex_lock");

  strcat(pamiec->napis, napis); // uwaga: możliwe przepełnienie bufora
  pamiec->napis_dostepny = 1;

  errno = pthread_cond_signal(&pamiec->cond);
  test_errno("pthread_cond_signal");

  errno = pthread_mutex_unlock(&pamiec->mutex);
  test_errno("pthread_mutex_unlock");

  printf("proces ustawił napis '%s' i wykonał pthread_cond_signal\n", napis
      );

  shmdt(pamiec);
  test_errno("shmdt");
}
//---------------------------------------------------------------------------

int main(int argc, char* argv[]) {
  if (argc > 2) {
    printf("proces 2: segment pamięci dzielonej %d\n", atoi(argv[1]));
    proces2(atoi(argv[1]), argv[2]);
  }
  else {
    puts("proces 1");
    proces1();
  }

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

## Przykład 18

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: pthread_atfork
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _XOPEN_SOURCE 700
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h> // sleep
#include <sys/wait.h> // waitpid

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

void* watek(void* numer) {
  printf("\turuchomiono wątek #%d\n", (int)numer);
  while (1) {
    printf("\t\twątek #%d w procesie #%d\n", (int)numer, getpid());
    usleep(700*1000);
  }

  return NULL;
}
//----------------------------------------------------------------------

#define N 3 /* liczba wątków */

pthread_t id[N];

void inicjalizacja_watkow() {
  int i;
  printf("tworzenie %d wątków w procesie %d\n", N, getpid());

  /* utworzenie wątków */
  for (i=0; i < N; i++) {
    errno = pthread_create(&id[i], NULL, watek, (void*)(i+1));
    test_errno("pthread_create");
  }
}
//----------------------------------------------------------------------

int main() {
  pid_t pid;
```

```
  puts("początek programu");
  inicjalizacja_watkow();

  /* rejestrowanie funkcji wykonywanej w procesie potomnym */
  errno = pthread_atfork(NULL, NULL, inicjalizacja_watkow);
  test_errno("pthread_atfork");

  sleep(1);

  pid = fork();
  printf("fork => %d\n", pid);
  switch (pid) {
    case -1:
      test_errno("fork");
      break;

    case 0: // proces potomny
      sleep(2);
      break;

    default: // proces nadrzędny
      waitpid(pid, NULL, 0);
      test_errno("waitpid");
      break;
  }

  /* kończymy proces, bez oglądania się na wątki */
  return EXIT_SUCCESS;
}
```

## Przykład 19

```
/*
 *  Przykładowy program dla kursu "POSIX Threads" z wikibooks.pl
 *
 *  Temat: priorytety wątków
 *
 *  Autor: Wojciech Muła
 *  Ostatnia zmiana: 2010-03-xx
 */

#define _XOPEN_SOURCE 500
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <errno.h>
#include <unistd.h>

#define test_errno(msg) do{if (errno) {perror(msg); exit(EXIT_FAILURE);}}
    while(0)

typedef struct {
  int licznik;
  char  przerwij;
  int priorytet;
} Arg;

/* funkcja wykonywana w wątku - zwiększa licznik */
void* watek(void* _arg) {
  Arg *arg = (Arg*)_arg;

  arg->licznik = 0;
  while (!arg->przerwij) {
    arg->licznik += 1;
    usleep(10);
  }

  return NULL;
}
//----------------------------------------------------------------------

#define N 4 /* liczba wątków */

int main(int argc, char* argv[]) {
  pthread_t id[N];
  pthread_attr_t attr;
  Arg arg[N];
  int pmin, pmax;
  int i, sched_policy;
  struct sched_param sp;

  sched_policy = SCHED_OTHER;
```

```
if (argc > 1)
  switch (atoi(argv[1])) {
    case 0:
      sched_policy = SCHED_OTHER;
      break;
    case 1:
      sched_policy = SCHED_RR;
      break;
    case 2:
      sched_policy = SCHED_FIFO;
      break;
  }
else {
  puts("program [0|1|2]");
  return EXIT_FAILURE;
}

pmin = sched_get_priority_min(sched_policy);
pmax = sched_get_priority_max(sched_policy);
switch (sched_policy) {
  case SCHED_OTHER:
    printf("SCHED_OTHER: priorytety w zakresie %d ... %d\n", pmin, pmax);
    break;
  case SCHED_RR:
    printf("SCHED_RR: priorytety w zakresie %d ... %d\n", pmin, pmax);
    break;
  case SCHED_FIFO:
    printf("SCHED_FIFO: priorytety w zakresie %d ... %d\n", pmin, pmax);
    break;
}

errno = pthread_attr_init(&attr);
test_errno("pthread_attr_init");

/* parametry szeregowania odczytywane z atrybutów  */
errno = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
test_errno("pthread_attr_setinheritsched");

/* wybór podanego algorytmu szeregowania */
errno = pthread_attr_setschedpolicy(&attr, sched_policy);
test_errno("pthread_attr_setschedpolicy");

/* utworzenie kilku wątków wątku z różnymi priorytetami */
for (i=0; i < N; i++) {
  /* kolejne wątki mają coraz wyższe priorytety */
  sp.sched_priority = pmin + (pmax-pmin) * i/(float)(N-1);
  arg[i].przerwij   = 0;
  arg[i].licznik    = 0;
  arg[i].priorytet  = sp.sched_priority;

  /* ustawienie priorytetu */
  errno = pthread_attr_setschedparam(&attr, &sp);
  test_errno("pthread_attr_setschedparam");
```

```
    /* uruchomienie wątku */
    errno = pthread_create(&id[i], &attr, watek, &arg[i]);
    test_errno("pthread_create");

    printf("utworzono wątek #%d o priorytecie %d\n", i, arg[i].priorytet);
  }

  errno = pthread_attr_destroy(&attr);
  test_errno("pthread_attr_destroy");

  /* oczekiwanie */
  sleep(2);

  /* ustawienie flagi zakończenia pracy, którą testują funkcje wątków
     oraz odczyt ich bieżących liczników */
  for (i=0; i < N; i++) {
    arg[i].przerwij = 1;
    printf("wątek #%d (priorytet %3d): licznik = %d\n",
      i,
      arg[i].priorytet,
      arg[i].licznik
    );
  }

  /* teraz oczekiwanie na ich zakończenie */
  for (i=0; i < N; i++) {
    errno = pthread_join(id[i], NULL);
    test_errno("pthread_join");
  }

  return EXIT_SUCCESS;
}
//---------------------------------------------------------------------------
```

# Funkcje

# Typy

# Stałe

# Definicje preprocesora