Wątki

1 Informacje wstępne 2
2 Rodzaje wątków 7
Wątki poziomu jądra 7
Wątki poziomu użytkownika 8
Rozwiązania mieszane 8
3 Biblioteka pthreads 12
Tworzenie i kończenie wątków 12
Przekazywanie danych do wątku i pobieranie wyników 20
Dostęp do wspólnych danych 24
Zmienne warunkowe 29
4 Blokady czytelników i pisarzy 41
5 Bariery 44
6 Wirujące blokady 47
7 Wątki w środowisku wieloprocesorowym 49

1 Informacje wstępne

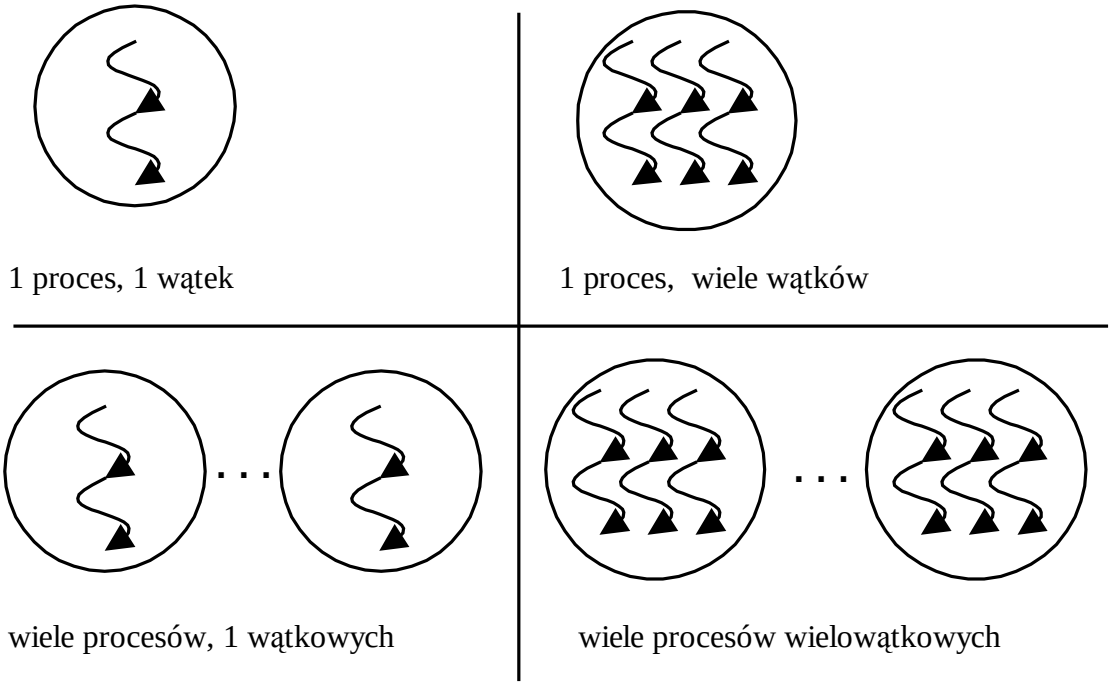
Wiele aplikacji ma do wykonania czynności które mogą być wykonane współbieżnie ale muszą dzielić dane i inne zasoby. Można zastosować procesy ale koszt komunikacji będzie znaczny. Lepszym rozwiązaniem jest abstrakcja wątku.

Tradycyjna implementacja procesu ma jeden wątek sterowania. W nowszych systemach pojęcie procesu zostało rozszerzone – dopuszcza się istnienie wielu wątków sterowania. Proces można podzielić na zbiór wątków i zestaw zasobów.

Proces – pojemnik na zasoby w ramach którego wykonują się watki.

Wątek – elementarna jednostka szeregowania korzystająca z zasobów procesu.

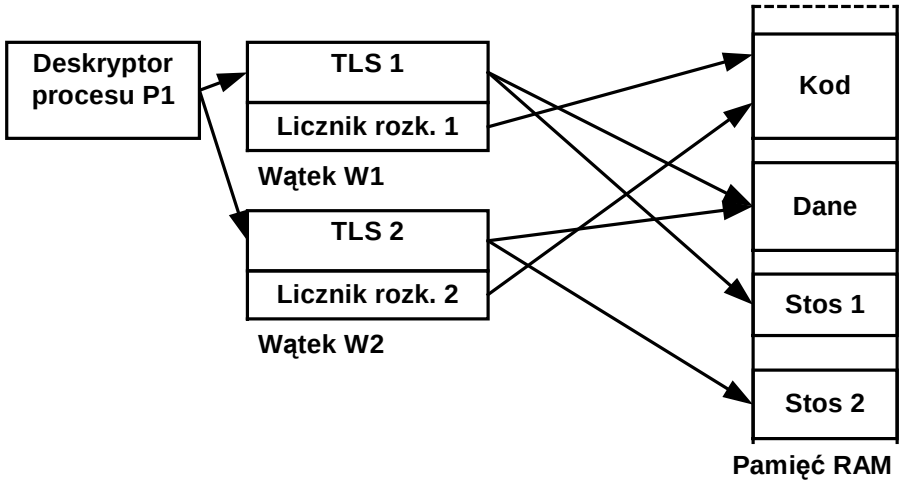
- Wątki wykonywane w ramach jednego procesu dzielą jego przestrzeń adresową i inne zasoby procesu.
- W ramach jednego procesu może się wykonywać wiele wątków



Rys. 1-1 Podstawowe modele przetwarzania

Atrybuty i zasoby własne wątku

1. Identyfikator wątka TID (*ang.Thread Identifier*) - każdy watek ma unikalny w ramach procesu identyfikator. Jest to liczba całkowita. Pierwszy wątek ma TID 1, następny 2 itd.
2. Zestaw rejestrów (*ang. Register set*) - każdy wątek posiada własny obszar pamięci w którym pamiętany jest zestaw rejestrów procesora (tak zwany kontekst procesora). Gdy watek jest wywłaszczany lub blokowany w obszarze tym pamiętane są rejestry procesora. Gdy watek będzie wznowiony obszar ten jest kopiowany do rejestrów procesora.
3. Stos *(ang. Stack*) - każdy wątek ma swój własny stos umieszczony w przestrzeni adresowej zawierającego go procesu. Na stosie tym pamiętane są zmienne lokalne wątku.
4. Maska sygnałów (*ang. Signal mask*) - każdy wątek ma swą własną maskę sygnałów. Maska sygnałów specyfikuje które sygnały mają być obsługiwane a które blokowane. Początkowa maska jest dziedziczona z procesu macierzystego.
5. Obszar TLS wątku (*ang. Thread Local Storage*) – każdy wątek ma pewien obszar pamięci przeznaczony na utrzymywanie różnych danych administracyjnych takich jak TID, PID, początek stosu, kod ewentualnego błędu **`errno`** i inne dane. Obszar TLS jest odpowiednikiem deskryptora procesu.
6. Procedura zakończenia (*ang. Cancellation Handler*) - gdy wątek się kończy wykonywana jest procedura zakończenia w ramach której zwalniane są zasoby wątku.



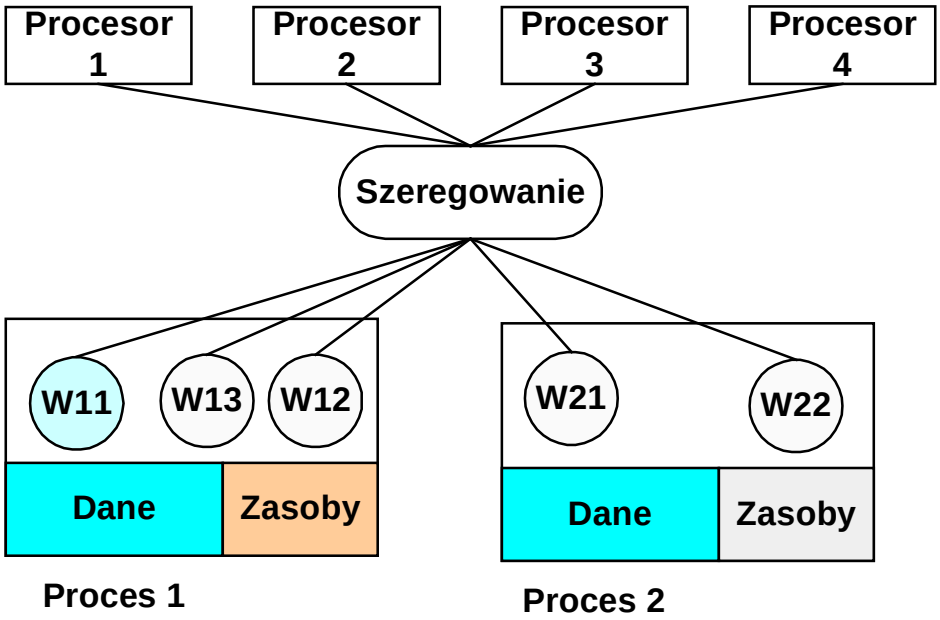
Rys. 1-2 Wątki 1 i 2 wykonywane w ramach procesu P1

Wątek dzieli ze swym procesem macierzystym następujące zasoby:
- Dane statyczne (segment danych)
- Deskryptory otwartych plików, blokady plików
- Maskę tworzenia plików (umask)
- Środowisko
- Katalog macierzysty i główny
- Limity zasobów (setrlimit)
- Timery
- Sesję, użytkownika, grupę, terminal sterujący

Zasoby własne wątku:
- Identyfikator wątka (thread ID)
- Maska sygnałów
- Zmienna errno
- Priorytet i strategię szeregowania

Rzeczywista równoległość

Wątki mogą być wykonywane na oddzielnych procesorach. Stąd model procesów wielowątkowych nadaje się do wykorzystania na multiprocesorach ze wspólną pamięcią (SMP – *Shared Memory Processors*) i zapewnia rzeczywistą równoległość przetwarzania.



Rys. 1-3 Procesy wielowątkowe wykonywane na wielu procesorach

Własności wątków

- Koszt utworzenia i przełączania wątku jest mniejszy niż procesu.
- Dane statyczne procesu są dla wątków działających w ramach jednego procesu wzajemnie widoczne.
- Wykonanie każdego wątku przebiega sekwencyjnie, każdy wątek ma swój licznik rozkazów.
- Wątki mogą być wykonywane na oddzielnych procesorach co umożliwia przyspieszenie obliczeń.
- Ponieważ wątki dzielą wspólne dane konieczna jest synchronizacja dostępu do tych wspólnych danych.

Zalety modelu wielowątkowego:

- Szybsza komunikacja pomiędzy wątkami wynikająca z mniej rygorystycznych mechanizmów kontroli.
- Mniejszy koszt tworzenia / kończenia wątku niż procesu
- Zwykle szybszy czas przełączania wątków niż procesów .
- Możliwość przetwarzania wieloprocesorowego SMP w systemach ze wspólną pamięcią (*ang. Symetrical Multi Procesing)*.

Wobec szybko malejących kosztów komputerów wieloprocesorowych ta ostatnia cecha nabiera szczególnie dużego znaczenia.

Zdefiniowanie działania wątków wymaga rozstrzygnięcia wielu kwestii.

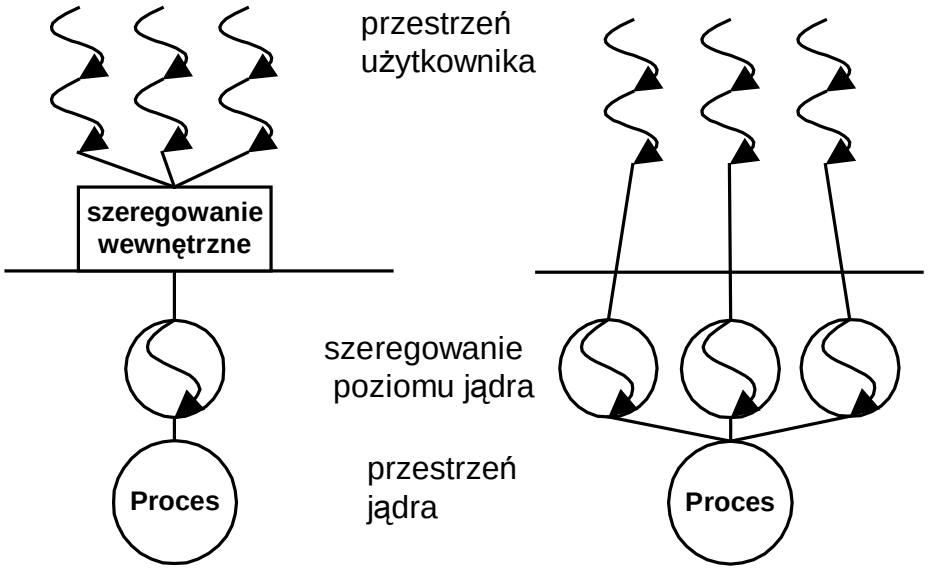
- Problem dzielenia dostępu do plików przez wiele wątków (wskaźnik bieżącej pozycji pliku, zamknięcie).
- Problem obsługi sygnałów trafiających do procesu (który wątek ma je obsługiwać)
- Problem widoczności wątków procesu przez inny proces (zwykle widoczność tylko w ramach procesu).

2 Rodzaje wątków

Klasyfikacja

Wyróżniane są następujące typy wątków:

- Wątki poziomu jądra KLT - (*ang. Kernel Level Tthreads*).
- Wątki poziomu użytkownika ULT - (*ang. User Level Threads*).
- Rozwiązania mieszane



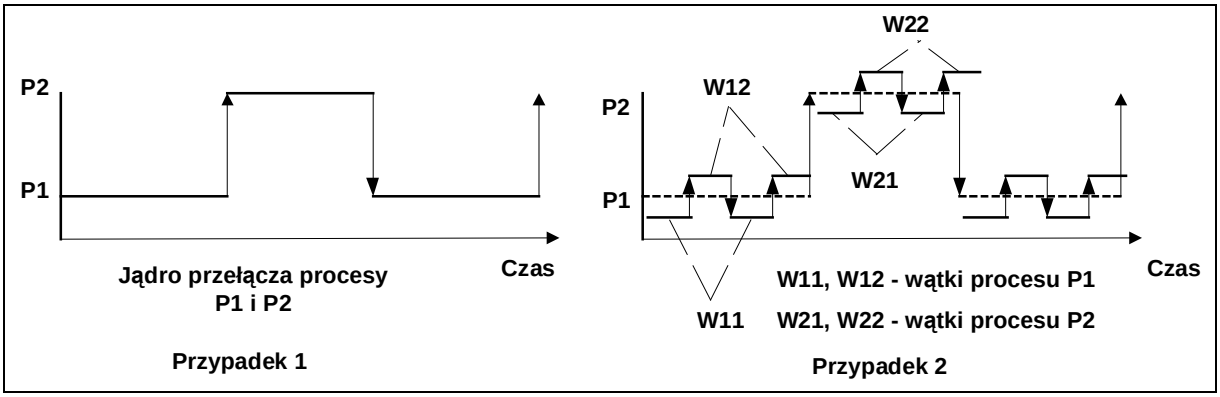
wątki poziomu użytkownika wątki poziomu jądra

Wątki poziomu jądra

Jądro jest świadome istnienia wątków i szereguje je niezależnie. Proces używający tych wątków w środowisku wieloprocesorowym może osiągnąć rzeczywistą równoległość.

Wątki poziomu użytkownika

Wątki poziomu użytkownika są tworzone bez wsparcia jądra. Kwanty czasu przydzielane danemu procesowi, są dzielone pomiędzy zawarte w nim wątki.
Każdy taki wątek ma swój własny stos, miejsce do pamiętania rejestrów i innych elementów kontekstu. Przełączanie kontekstów pomiędzy wątkami procesu odbywa się bez udziału jądra przez specjalne biblioteki (np. pthreads).

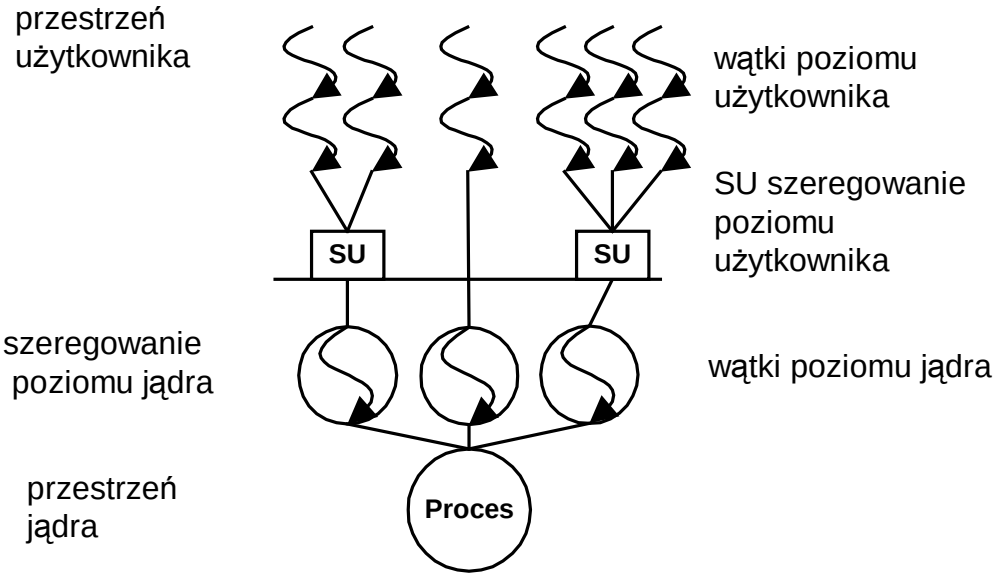


Przypadek 1 – Jądro przydziela czas procesora procesom P1 i P2

Przypadek 2 – Wątki W11 i W12 wykonywane w ramach czasu przydzielonego procesowi P1 przez jądro. Podobnie W21 i W22. W ramach jednego procesu przełączanie wykonywane jest przez bibliotekę użytkownika.

Rozwiązania mieszane

Istnieją także rozwiązania pośrednie pomiędzy wątkami jądra a wątkami użytkownika. Są to procesy lekkie (*ang. Lightweighted processes*) systemu Solaris. W rozwiązaniu tym w jednym wątku poziomu jądra może istnieć kilka wątków poziomu użytkownika.



Zalety wątków użytkownika:

- Szybkość działania
- Zwiększenie stopnia współbieżności aplikacji.

Wady wątków użytkownika:

- Wywołanie funkcji blokującej proces wewnątrz wątku , wstrzymuje inne wątki.
- Brak możliwości zrównoleglenia (wykonania na wielu procesorach).

Mniejsza szybkość wątków jądra wynika z konieczności przełączenia trybu *user / kernel* w przypadku wykonania funkcji systemowych.

Wyniki testów:
Zastosowanie wątków poziomu użytkownika przyspiesza przełączanie o ok. rząd wielkości.

Null fork – utworzenie i zakończenie pustego procesu
Signal – Wait – wysłanie sygnału do innego wątku i oczekiwanie na odpowiedź.

	Wątki poziomu użytkownika	Wątki poziomu jądra	Procesy
Null fork	34	948	11300
Signal - Wait	37	441	1840

Czasy w MS maszyna VAX system UNIX

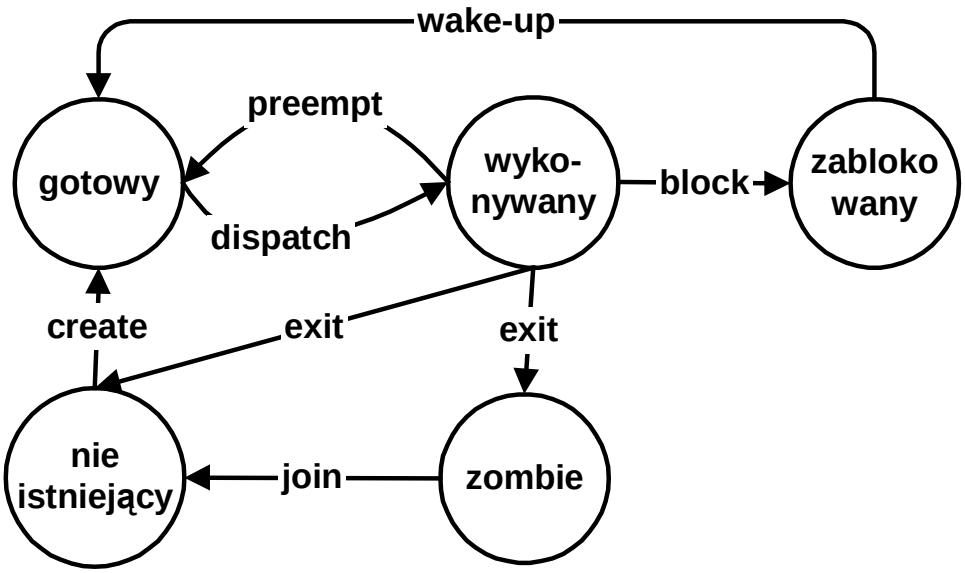
Kiedy opłacalne jest stosowanie wątków zależy od udziału operacji przełączania kontekstu w całości. Zależy od natury aplikacji.

Zestaw funkcji interfejsowych niezbędnych do operowanie na wątkach: Aby umożliwić operowanie na wątkach niezbędny jest pewien zestaw funkcji interfejsowych. Powinny być udostępnione następujące grupy funkcji:

- Tworzenie wątków i ich kasowanie.
- Synchronizacja dostępu do wspólnych danych (muteksy).
- Synchronizacja działania wątków (zmienne warunkowe, bariery).
- Operowanie atrybutami wątków (priorytet, protokół unikania inwersji priorytetów, maska sygnałów).

Kanoniczne stany wątku

Wykonywany	Właśnie wykonywany przez procesor
Gotowy	Posiadający wszyskie zasoby oprócz procesora
Zablokowany	Brak pewnych zasobów
Zombie	Zakończony ale nie dołączony do wątku macierzystego (wątek dołączalny)
Nieistniejący	Nie utworzony lub zakończony



Rys. 2-1 Diagram stanów kanonicznych wątków

3 Biblioteka pthreads

Zestaw funkcji dotyczący wątków zdefiniowany został przez normę POSIX P1003.4a i nosi nazwę pthreads (skrót od POSIX threads). Implementacja pakietu istnieje między innymi w systemie Linux, QNX6, DEC OSF1. Obecnie wątki są elementem biblioteki glibc (Od wersji 2).

W systemie Linux występuja dwie biblioteki dotyczące wątków:

- LinuxThreads – poczatkowa biblioteka wątków Linuxa
- NPTL (Native POSIX Threads Library) - bardziej zgodna ze standardem POSIX, wspierana od jądra 2.6

Podstawowe operacje na wątkach:

Tworzenie i kończenie	Tworzenie i kasowanie wątków, czekanie na zakończenie, inicjowanie zakończenia.
Testowanie i ustawianie atrybutów wątków	Priorytet, strategia szeregowania, wielkość stosu, maska skojarzeń, protokół unikanie inwersji priorytetów
Operowanie na muteksach	Zabezpieczenie dostępu do wspólnych danych. Tworzenie i kasowanie muteksów, zajmowanie i zwalnianie muteksów.
Operowanie na zmiennych warunkowych	Używane do synchronizacji wątków. Tworzenie i kasowanie zmiennych warunkowych, zawieszanie i wznawianie wątków.
Operowanie na barierach	Używane do synchronizacji wątków. Tworzenie bariery, czekanie na barierze

Tworzenie i kończenie wątków

`pthread_create`	Tworzenie wątku
`pthread_exit`	Zakończenie wątku bieżącego
`pthread_join`	Czekanie na zakończenie wątku
`pthread_attr_init`	Inicjacja atrybutów wątku
`pthread_self`	Pobranie identyfikatora wątku
`pthread_yield`	Zwolnienie procesora
`pthread_cancel`	Kasowanie innego wątku

Tworzenie wątku

Nowy wątek tworzy się przy pomocy funkcji **pthread_create.** Funkcja ta tworzy wątek, którego kod znajduje się w funkcji podanej jako argument **func**. Wątek jest uruchamiany z parametrem **arg**, a informacja o nim jest umieszczana w strukturze **thread**.

```
int pthread_create( pthread_t *thread,
pthread_attr_t *attr,void (* func)(void *), void
*arg)
```

thread	identyfikator wątku – wartość nadawana przez funkcję
attr	atrybuty wątku, gdy NULL przyjęte domyślne
func	procedura zawierająca kod wątku
arg	argument przekazywany do wątku

Funkcja zwraca: 0 – sukces, -1 – błąd.

```
#include <pthread.h>
#include <stdio.h>

void * kod(void *arg) {
   while(1) {
      putc('W',stderr);
      sleep(1);
   }
   return (NULL);
}

int main(int argc, char *argv[]) {
  int tid;
  pthread_create(&tid, NULL, kod,NULL);
  while(1) {
     putc('M',stderr);
     sleep(1);
  }
  return  0;
}
```

Przykład 3-1 Tworzenie wątku za pomocą funkcji **pthread_create**

```
$./thread1
MWMWMWMWMWMWMWMWMWMWMWMWMWMWMWMW
```

Wynik 3-1 Działanie programu **thread1**

Kończenie wątku

Wątek może być zakończony w następujące sposoby:

- Następuje powrót z procedury określającej kod wątku.
- Wątek wykonuje funkcję **pthread_exit()**.
- Wątek jest kasowany przez inny wątek.
- Następuje zakończenie procesu macierzystego wątku.

Jawne zakończenie wątku następuje poprzez wywołanie procedury:

```
pthread_exit(void * status)
```

status Kod powrotu wątku

Możliwe są dwa sposoby postępowania z kończonymi wątkami:

1. Z chwilą zakończenia się wątku zwalniane są wszystkie jego zasoby.
2. Zasoby zwalniane są z chwilą dołączenia wątku bieżącego do innego wątku (wykonującego funkcję **pthread_join**).

Postępowanie to uzależnione jest od ustawienia atrybutu **PTHREAD_CREATE_JOINABLE** który ustalany jest podczas tworzenia wątku.

1. Gdy atrybut ten nie jest ustawiony, wątek zwalnia swe zasoby zaraz po zakończeniu.
2. Gdy atrybut jest ustawiony, wątek zwalnia zasoby po dołączeniu do innego wątku.

Oczekiwanie na zakończenie wątku.

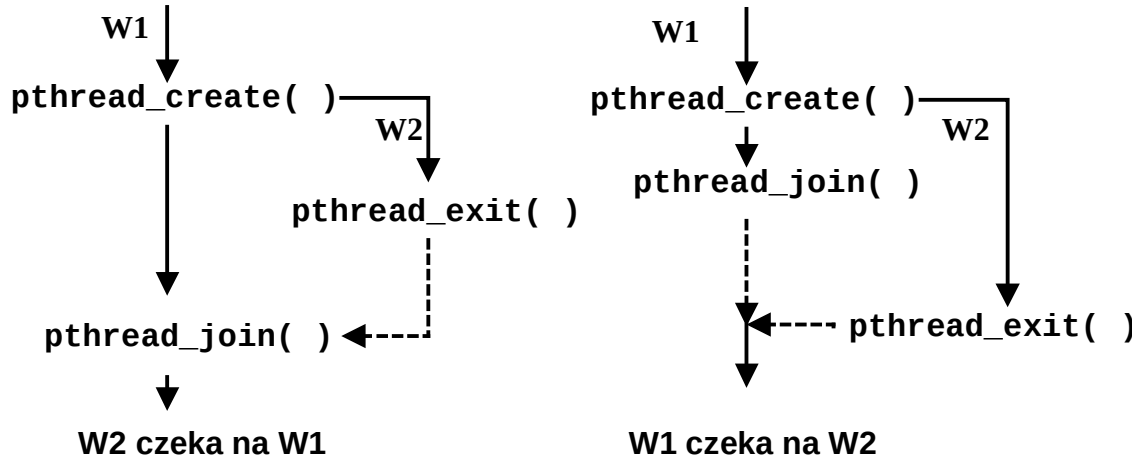
Proces bieżący może czekać na zakończenie innego wątku poprzez wywołanie funkcji **pthread_join**.

```
int pthread_join( pthread_t *thread, void *status)
```

thread identyfikator wątku – wartość nadawana przez funkcję
status Kod powrotu zwracany przez zakończony wątek

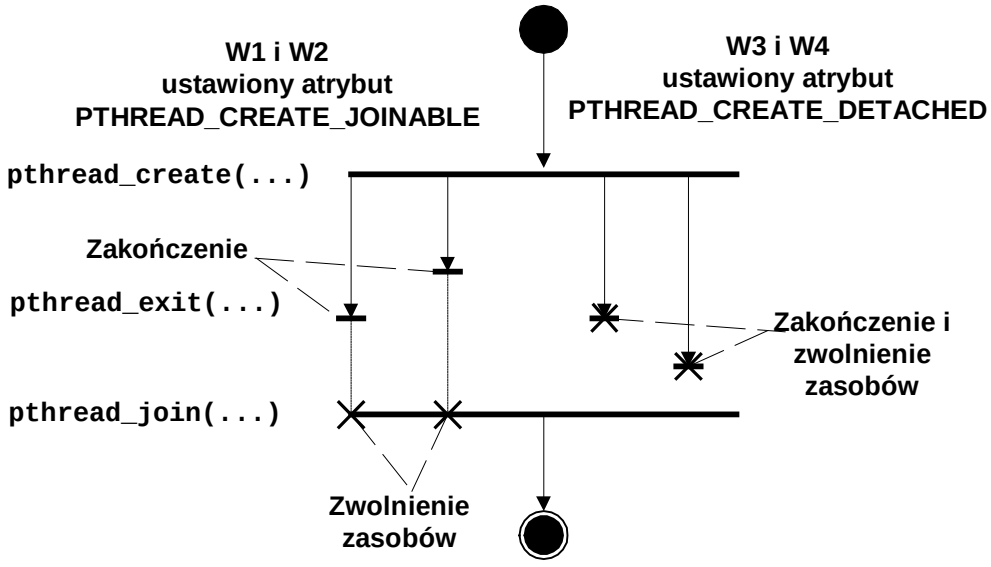
Gdy wskazany jako parametr wątek nie zakończył się jeszcze, wątek bieżący jest wstrzymywany.

Funkcja zwraca: 0 – sukces, -1 – błąd.



Rys. 3-1 Zakończenia wątków

Sposób kończenia wątku zależny jest od atrybutu **PTHREAD_CREATE_JOINABLE.** Według standardu POSIX nowo tworzone wątki domyślnie mają ustawiony ten atrybut a więc są dołączalne.



Rys. 3-2 Wątki dołączalne i nie dołączalne

Atrybuty wątku

Atrybuty wątku są to jego dodatkowe własności jak dołączalność, priorytet, strategia szeregowania, strategia unikania inwersji priorytetów, rozmiar stosu.
Atrybuty są przekazywane do wątku w chwili jego tworzenia jako wartość parametru attr funkcji pthread_create.

Aby zmienić atrybuty wątku należy:

1. Zadeklarować zmienną **`attr`** typu **`pthread_attr_t`** .
2. Zainicjować zmienną za pomocą funkcji **`pthread_attr_init(&attr)`**
3. Zmodyfikować strukturę zawierającą atrybuty tak aby atrybuty miały pożądaną wartość.
4. Wywołać funkcję **`pthread_create(…,&attr`**) tworzącą nowy wątek i przekazać jej wskaźnik na strukturę zawierającą atrybuty.
5. Zwolnić pamięć zajmowaną przez atrybut poprzez wykonanie funkcji **`pthread_attr_destroy (&attr).`**

`pthread_init` – inicjacja atrybutów wątku

```
int pthread_init(pthread_attr_t *attr)
```

`attr` Wskaźnik do inicjowanych atrybutów

Funkcja przydziela pamięć na atrybuty i nadaje im wartości domyślne. Struktura zawierająca atrybuty może być wykorzystana przy tworzeniu wielu wątków. Ważniejsze atrybuty wątku są następujące:

1. Dołączalność – informacja czy po zakończeniu wątek ma zwolnić zasoby natychmiast czy dopiero gdy wątek macierzysty wykona funkcję **`pthread_join()`**.
2. Strategia szeregowania – **`SCHED_FIFO, SCHED_RR, SCHED_OTHER, SCHED_NOCHANGE, SCHED_SPORADIC`**.
3. Parametry szeregowania – różne informację (między innymi priorytet) używanie do szeregowania wątku.

4. Rozmiar stosu – informacja jaki ma być rozmiar stosu. Domyślna wartość wynosi 4 KB (może być zmieniona przez funkcję **`pthread_attr_setstacksize).`**
5. Adres stosu – początkowy adres stosu lub wartość NULL. Gdy parametr ma wartość NULL adres stosu będzie ustalany automatycznie przez system. Pamięć na stos może być przydzielona przez programistę (musi wtedy sam go zwolnić).

Atrybut	Wartość domyślna
Dołączalność	**`PTHREAD_CREATE_JOINABLE`**
Strategia szeregowania	**`PTHREAD_INHERID_SCHED`**
Parametry szeregowania	Dziedziczone z procesu macierzystego
Rozmiar stosu	4 KB
Adres stosu	**`NULL`**

Tabela 3-1 Atrybuty wątku i ich wartości domyślne

Atrybut	Funkcja testowania	Funkcja ustawiania
Dołączalność	**`attr_getdetachstate()`**	**`attr_settdetachstate()`**
Strategia szeregowania	**`attr_getschedpolicy()`**	**`attr_setschedpolicy()`**
Parametry szeregowania	**`attr_getschedparam()`**	**`attr_setschedparam()`**
Rozmiar stosu	**`attr_getstacksize(), attr_getstacklazy()`**	**`attr_setstacksize(), attr_setstacklazy()`**
Adres stosu	**`attr_getstackaddr()`**	**`attr_setstackaddr()`**

Tabela 3-2 Ważniejsze funkcje do ustalania atrybutów wątku

```
void *kod(void *data){
  ....
  return (void *) cos;
}

main(void) {
 pthread_t thr;
 pthread_attr_t attr;
 int status;
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
 pthread_create(&thr, &attr, kod,0);
 ....
 pthread_join(thr,(void *) &status);
 pthread_attr_destroy(&attr)
}
```

Przykład 3-2 Ustawiania atrybutów wątku, utworzenia i oczekiwania na zakończenie

Uzyskanie własnego identyfikatora

Wątek może uzyskać własny identyfikator poprzez wywołanie funkcji:

```
pthread_t pthread_self(void)
```

Zwolnienie procesora

Wywołanie funkcji **`pthread_yield`** powoduje przekazanie sterowania do procedury szeregującej która wybierze następny wątek do wykonania.

```
int pthread_yield(void)
```

```
/* Kompilacja: cc thread1.c -o thread1 -lpthread    */
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 6
#define KROKOW      16

pthread_t tid[NUM_THREADS]; // Tablica identyfik. watkow

void * kod(void *arg);

int main(int argc, char *argv[]) {
   int i;
   // Tworzenie watkow -------------
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i], NULL, kod,(void *)(i+1));

   // Czekanie na zakonczenie ------
   for (i = 0; i < NUM_THREADS; i++)
              pthread_join(tid[i], NULL);
   printf("watki zakonczone \n");
   return  0;
}

void * kod(void *arg) {
   int numer = (int)arg;
   int i;
   printf("watek: %d numer: %d \n", getpid(), numer);
   for(i=0;i<KROKOW;i++) {
    printf("watek %d krok %d \n",numer,i);
    sleep(1);
   }
   return (NULL);
}
```

Przykład 3-3 Tworzenie, wykonanie i kończenie wątków

Przekazywanie danych do wątku i pobieranie wyników

Parametry wątku

Funkcja realizująca wątek ma tylko jeden parametr będący wskaźnikiem do typu void.

```
void * kod(void *arg)
```

Aby przekazać do wątku większą niż jeden liczbę parametrów należy się posłużyć:

- strukturą
- tablicą.

```
typedef struct {
     int pocz;
     int kon;
     int numer;
} param_t;
```

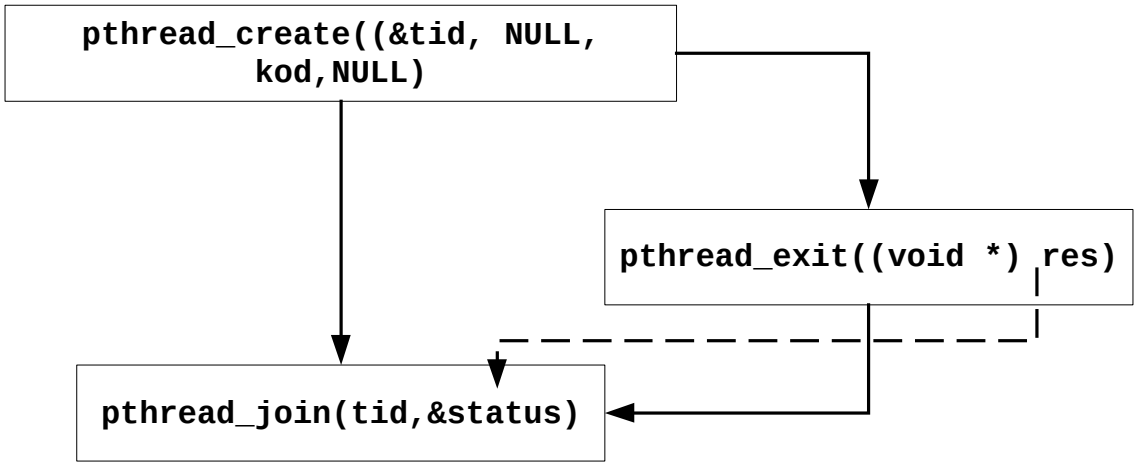
Wyniki wątku

Wątek powinien zwracać wskaźnik na void poprzez:

- Wykonanie **`return res`**
- Wykonanie funkcji **`pthread_exit(void *res)`**

Wartość ta może być odczytana w funkcji:

```
int pthread_join( pthread_t *thread, void *status)
```



```
typedef struct {
     int pocz;
     int kon;
     int numer;
} param_t;

int main(int argc, char *argv[]){
  param_t arg;
  int wynik;
  ...
  for (i = 0; i < NUM_THREADS; i++) {
    arg.pocz = ...;
    arg.kon  = ...;
    arg.numer = i;
    pthread_create(&tid[i], NULL, znajdz,&arg);
  }
  ...
// Czekanie na zakonczenie ------
   for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], (void*)&wynik);
        printf("Watek: %d wynik: %d\n",i,wynik);
   }
}

void *znajdz(void *arg) {
 int x1,x2,num,res;
 param_t *par;
 par = (param_t *)arg;
 x1 = par->pocz;
 x2 = par->kon;
 num = par->numer;
 printf("watek: %d pocz: %d kon: %d \n",num,x1,x2);
 ...
 res = ...;
 return((void *) res);
}
```

Przykład 3-4 Przekazywanie parametrów do wątku i uzyskiwania wyników

Anulowanie wątku

Normalnym sposobem zakończenia się wątku jest:
1. wykonanie instrukcji **`return`** lub
2. funkcji **`pthread_exit`**.

Wtedy wątek sam podejmuje decyzję o swym zakończeniu.

Możliwe jest jednak aby jeden wątek zakończył inny. Używana jest do tego celu funkcja **`pthread_cancel`**.

```
int pthread_cancel(pthread_t *thread)
```

`thread` Identyfikator kasowanego wątku

Należy unikać anulowania wątków przez inne watki.

Powodem jest fakt że wątek mógł pobrać pewne zasoby systemowe (pamięć, pliki) i rozpocząć pewne akcje synchronizacyjne (np. zająć muteks). Gdy zostanie on zakończony w przypadkowym punkcie wykonania może nie zwrócić pobranych zasobów ani też nie zakończyć rozpoczętych akcji synchronizacyjnych. Skutkować to może wyciekaniem zasobów lub wręcz blokadą aplikacji.

Procedura czyszcząca - funkcja która będzie wykonana automatycznie gdy wątek będzie anulowany i jej zadaniem jest zwolnienie pobranych przez wątek zasobów.

Funkcja czyszcząca jest aktywowana poprzez wykonanie funkcji **`pthread_cleanup_push(...)`**

Funkcja czyszcząca jest deaktywowana poprzez funkcję **`pthread_cleanup_pop(...)`** lub poprzez normalne zakończenie się wątku.

Dostęp do wspólnych danych

Wątki dzielą wspólny obszar danych. Stąd współbieżny dostęp do danych może naruszyć ich integralność. Należy zapewnić synchronizację dostępu do wspólnych danych. W bibliotece pthreads do zapewnienia wyłączności dostępu do danych stosuje się mechanizm muteksu (*ang. mutex*). Nazwa ta pochodzi od słów *Mutual exclusion* czyli wzajemne wykluczanie.

pthread_mutex_init	Inicjacja muteksu
pthread_mutex_lock	Zajęcie muteksu
pthread mutex timedlock	Zajęcie muteksu – czekanie ograniczone czasowo
pthread_mutex_unlock	Zwolnienie muteksu
pthread_mutex_destroy	Skasowanie muteksu

Deklaracja muteksu

Muteks jest obiektem abstrakcyjnym który może być w dwu stanach: wolny i zajęty. Na muteksie wykonuje się dwie podstawowe operacje: zajęcie i zwolnienie.

Biblioteka pthreads definiuje muteks jako typ **pthread_mutex_t.** Przed użyciem obiekt typu muteks musi być zadeklarowany. Przykładowo muteks o nazwie blokada deklaruje się jak poniżej.

```
pthread_mutex_t blokada;
```

Inicjacja muteksu

Przed użyciem muteks musi być zainicjowany. Inicjacja następuje poprzez wykonanie funkcji:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr)
```

mutex Zadeklarowana wcześniej zmienna typu **pthread_mutex_t.**

attr Atrybuty muteksu. Gdy **attr** jest równe NULL przyjęte będą wartości domyślne.

Funkcja zwraca: 0 – sukces, -1 – błąd.
Zainicjowany muteks pozostaje w stanie odblokowania.

Zablokowanie dostępu do zasobu

Przed dostępem do zasobu należy zapewnić sobie wyłączność w korzystaniu z tego zasobu. W tym celu wątek wykonuje funkcję:

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

mutex Zadeklarowana wcześniej i zainicjowana zmienna typu **pthread_mutex_t**

Działanie funkcji zależy od stanu w jakim znajduje się muteks.

1. Gdy muteks jest wolny, następuje jego zablokowanie.
2. Gdy muteks jest zajęty, próba jego powtórnego zajęcia powoduje zablokowanie się wątku który tę próbę podejmuje.

Ograniczone czasowo blokowanie zasobu

```
int pthread mutex timedlock(pthread mutex t * mutex,
struct timespec * timeout );
```

mutex Zadeklarowana wcześniej i zainicjowana zmienna typu **pthread_mutex_t**

timeout Okres oczekiwania

```
struct timespec {
     time_t tv_sec;
     time_t tv_nsec;
}
```

Gdy muteks nie jest wolny watek się blokuje ale po upływie zadanego okresu ulega odblokowaniu.

Odblokowanie dostępu do zasobu

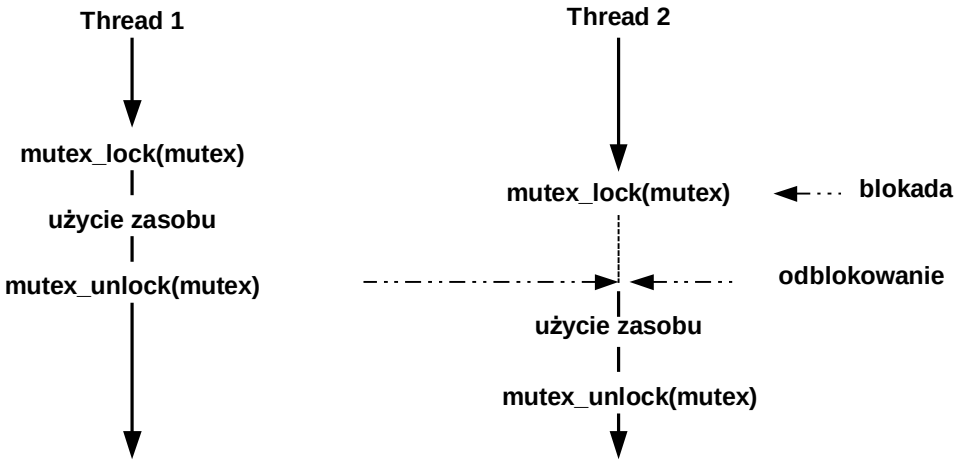
Użyty i zablokowany wcześniej zasób powinien być zwolniony. Zwolnienie zasobu odbywa się poprzez wywołanie funkcji:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

mutex Zadeklarowana wcześniej i zainicjowana zmienna typu **pthread_mutex_t**

Działanie funkcji zależy od tego czy inne wątki czekają zablokowane na muteksie.

1. Brak wątków zablokowanych na muteksie – stan muteksu zostaje zmieniony na wolny.
2. Są wątki zablokowane na muteksie – jeden z czekających wątków zostaje odblokowany.



Rys. 3-3 Dwa wątki używają wspólnego zasobu chronionego przez mutex.

Kasowanie muteksu

Kasowanie muteksu odbywa się poprzez wywołanie funkcji:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

System zwalnia zasoby zajęte przez muteks. Muteks musi być wolny gdyż w przeciwnym razie nie będzie skasowany i funkcja zwróci kod błedu.

```
pthread_t wri;
int  bufor[SIZE];
int  new_item = 0;
int  step = 0;

void *writer(void *arg);
void  reader(void );

pthread_mutex_t mutex;

int main(int argc, char *argv[]) {
   int i;
   pthread_mutex_init(&mutex,NULL);
   pthread_create(&wri, NULL, writer,NULL);
   reader();
   pthread_join(wri, NULL);
   printf("watki zakonczone \n");
   return  0;
}  /* main */

// Watek piszacy do bufora
void * writer(void *arg){
   int i;
   while(1) {
    pthread_mutex_lock(&mutex);
    // while(new_item == 1);
    if(new_item == 0) {
        pisz(bufor);
        new_item = 1;
    }
    pthread_mutex_unlock(&mutex);
   }
   return (NULL);
}

// Wątek czytający z bufora
```

```
void reader(void ) {
   int i;
   while(1) {
    pthread_mutex_lock(&mutex);
    // while(new_item == 0);
    if(new_item == 1) {
        czytaj(bufor)
        new_item = 0;
    }
    pthread_mutex_unlock(&mutex);
   }
}
```

Przykład 3-5 Problem producenta / konsumenta – rozwiązanie z odpytywaniem

Zmienne warunkowe

Muteksy są narzędziem zapewniającym ochronę sekcji krytycznej. Jak postąpić gdy wewnątrz sekcji krytycznej wymagane jest oczekiwanie na spełnienie pewnego warunku a więc synchronizacja z innym wątkiem. Podobny problem występuje w monitorze gdzie używa się funkcji **`wait`**(...) i **`signal`**(...) . Biblioteka wątków dostarcza narzędzia do rozwiązania takiego problemu synchronizacji. Narzędzie to nazywa się zmienną warunkową (*ang. condition variable*).

Zmienna warunkowa jest narzędziem do blokowania wątku wewnątrz sekcji krytycznej aż do momentu gdy pewien warunek zostanie spełniony. Warunek ten może być dowolny i niezależny od zmiennej warunkowej. Zmienna warunkowa musi być użyta w połączeniu z muteksem o ile konstrukcja ma zapewnić własności monitora.

Przed użyciem zmienna warunkowa musi być zadeklarowana jako zmienna typu **`pthreads_cond_t.`**

Najważniejsze operacje związane ze zmiennymi warunkowymi są dane poniżej.

`pthread_cond_init`	Inicjacja zmiennej warunkowej.
`pthread_cond_wait`	Czekanie na zmiennej warunkowej
`pthread_cond_timedwait`	Ograniczone czasowo czekanie na zmiennej warunkowej
`pthread_cond_signal`	Wznowienie wątku zawieszonego w kolejce danej zmiennej warunkowej.
`pthread_cond_brodcast`	Wznowienie wszystkich wątków zawieszonych w kolejce danej zmiennej warunkowej.
`pthread_cond_destroy`	Skasowanie zmiennej warunkowej z zwolniene jej zasobów.

Inicjacja zmiennej warunkowej

```
int pthread_cond_init(pthreads_cond_t *zw,
pthreads_condattr_t attr)
```

zw	Zadeklarowana wcześniej zmienna typu **pthread_cond_t.**
attr	Atrybuty zmiennej warunkowej. Gdy **attr** jest równe NULL przyjęte będą wartości domyślne.

Zawieszenie wątku w oczekiwaniu na sygnalizację

```
int pthread_cond_wait(pthreads_cond_t *zw,
pthread_mutex_t *mutex)
```

zw	Zadeklarowana i zainicjowana zmienna typu **pthread_cond_t.**
mutex	Zadeklarowana i zainicjowana zmienna typu **pthread_mutex_t.**

Funkcja powoduje zawieszenie bieżącego wątku w kolejce związanej ze zmienną warunkową zw. Jednocześnie blokada mutex zostaje zwolniona. Obie operacje są wykonane w sposób atomowy.
Gdy inny wątek wykona operację **pthreads_cond_signal(&zw)** zablokowany wątek zostanie odblokowany a blokada mutex zwolniona.

Zawieszenie wątku w oczekiwaniu na sygnalizację z ograniczeniem czasowym

```
int pthread_cond_timedwait(pthreads_cond_t *zw,
pthread_mutex_t *mutex,struct timespec *abstime)
```

zw	Zadeklarowana i zainicjowana zmienna typu **pthread_cond_t.**
mutex	Zadeklarowana i zainicjowana zmienna typu **pthread_mutex_t.**
abstime	Timeout oczekiwania, czas absolutny

Funkcja powoduje zawieszenie bieżącego wątku w kolejce związanej ze zmienną warunkową zw. Jednocześnie blokada mutex zostaje zwolniona. Obie operacje są wykonane w sposób atomowy.
Gdy inny wątek wykona operację **`pthreads_cond_signal(&zw)`** zablokowany wątek zostanie odblokowany a blokada mutex zwolniona.
Gdy przekroczony zostanie czas oczekiwania watek będzie odblokowany.

Wznowienie zawieszonego wątku

```
int pthread_cond_signal(pthreads_cond_t *zw)
```

`zw` Zadeklarowana i zainicjowana zmienna typu **`pthread_cond_t.`**

Jeden z wątków zablokowanych na zmiennej warunkowej zw zostanie zwolniony.

Wznowienie wszystkich zawieszonych wątków

```
int pthread_cond_brodcast(pthreads_cond_t *zw)
```

`zw` Zadeklarowana i zainicjowana zmienna typu **`pthread_cond_t.`**

Wszystkie wątki zablokowane na zmiennej warunkowej zw zostaną zwolnione.

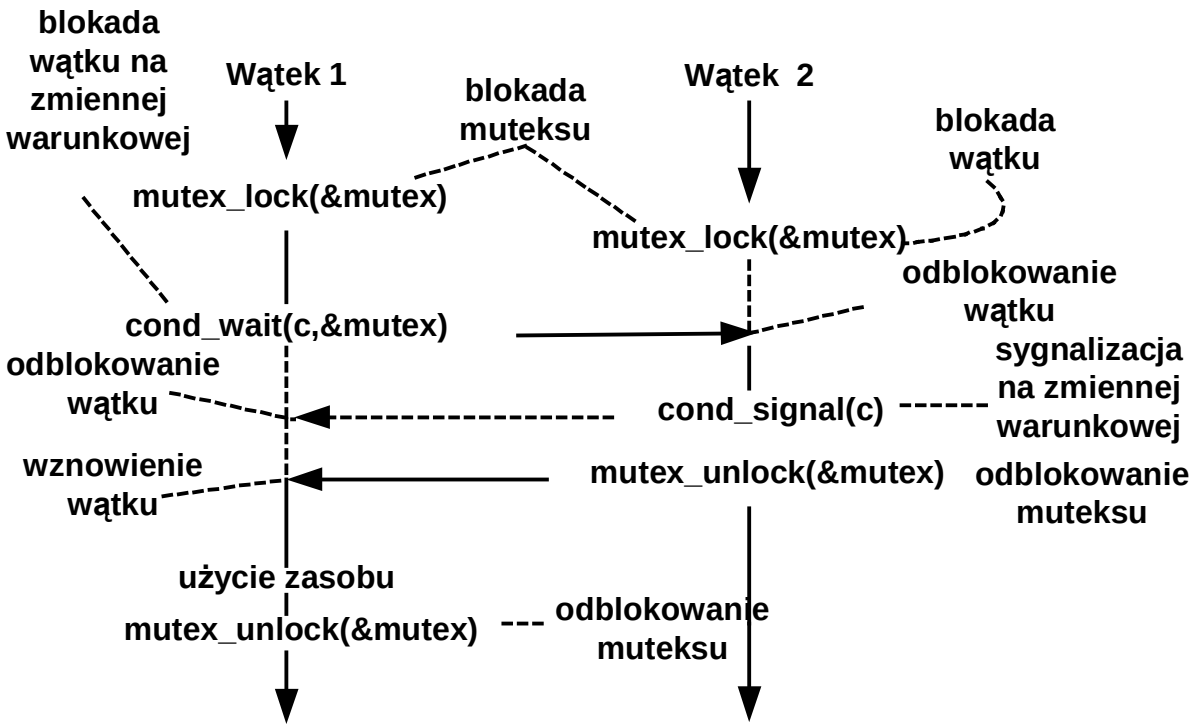
Typowa sekwencja użycia zmiennej warunkowej:

```
pthread_mutex_lock(&m)
...
while( ! warunek )
    pthread_cond_wait( &cond, &m)
...
pthread_mutex_unlock(&m)
```

Wątek oczekujący na warunek

```
pthread_mutex_lock(&m)
...
ustawienie_warunku
pthread_cond_signal( &cond)
...
pthread_mutex_unlock(&m)
```

Wątek ustawiający warunek i sygnalizujący jego spełnienie



Rys. 3-4 Wątek 1 czeka na warunek ustawiany przez wątek 2

```
// Problem czytelników i pisarzy rozw. poprawne
pthread_mutex_t mutex;   // Ochrona zmiennej count
pthread_cond_t cond;     // Kolejka czekających
// Liczba czyt. w czytelni, -1 gdy jest tam pisarz
int count;

read_lock(void) {
  mutex_lock(&mutex);
  // Czytelnik czeka gdy w czytelni jest pisarz
  while(count < 0)
     pthread_cond_wait(&cond,&mutex);
  count++;
  mutex_unlock(&mutex);
}

write_lock(void) {
  mutex_lock(&mutex);
  // Pisarz czeka na wolną czytelnie
  while(count != 0)
     pthread_cond_wait(&cond,&mutex);
  count--;
  mutex_unlock(&mutex);
}

rw_unlock(void) {
  mutex_lock(&mutex);
  // Pisarz czeka na wolną czytelnie
  if(count < 0)   count = 0
  else            count --;
  if(count == 0)
     cond_brodcast(&cond);
  mutex_unlock(&mutex);
}

void pisarz(void) {
  while(1) {
   rw_rdlock();
   czytanie(...);
   rw_unlock();
  }
}
```

```
void czytelnik(void) {
  while(1) {
   rw_wrlock();
   pisanie(...);
   rw_unlock();
  }
}
```

Przykład 3-6 Rozwiązanie problemu czytelników i pisarzy

Rozwiązanie problemu producenta - konsumenta

```
#include <stdio.h>
#include <pthread.h>
#define N 4
int pocz ,kon,licznik = 0;
int bufor[N];
pthread_mutex_t mutex;
pthread_cond_t puste, pelne;

void* producent( void* arg ) {
  int num = 0;
  int cnt = 1;
  num = (int) arg;
  printf("Start producent: %d\n",num);
  while( 1 ) {
    pthread_mutex_lock( &mutex );
    while(licznik >= N)
            pthread_cond_wait(&puste,&mutex);
    bufor[kon] = cnt++;
    kon = (kon+1) %N;
    licznik++;
    pthread_cond_signal(&pelne);
    pthread_mutex_unlock( &mutex );
    printf("Prod%d kon: %d wst: %d\n", num, kon,cnt,
            licznik );
    sleep( 1 );
  }
}

void* konsument( void* arg ) {
  int num,x = 0;
  num = (int) arg;
  printf("Start konsument: %d\n",num);
  while( 1 ) {
    pthread_mutex_lock( &mutex );
    while(licznik <=0 )
          pthread_cond_wait(&pelne,&mutex);
    x = bufor[pocz];
    pocz = (pocz+1) %N;
    licznik--;
    pthread_cond_signal(&puste);
    pthread_mutex_unlock( &mutex );
    printf("Kons%d pocz: %d pobral: %d licz: %d\n",
              num, pocz, x,licznik );
    sleep( 1 );
  }
}
```

```
int main( void ) {
   pthread_t p1,p2,k1,k2;
   pthread_mutex_init(&mutex,NULL);
   pthread_cond_init(&puste,NULL);
   pthread_cond_init(&pelne,NULL);
   pthread_create(&p1, NULL, &producent, (void *)1 );
   pthread_create(&k1, NULL, &konsument, (void *)1 );
   pthread_join(p1,NULL);
   pthread_join(k1,NULL);
   return 0;
}
```

Przykład 3-7 Problem producenta i konsumenta - rozwiązanie za pomocą zmiennych warunkowych

Przykład – Znajdowanie liczb pierwszych

```
//-------------------------------------------------------------
// Znajdowanie liczb pierwszych  program wielowatkowy
// Uruchomienie: ./pierwsze  zakres_od zakres_do watkow
//-------------------------------------------------------------
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 8

pthread_t tid[NUM_THREADS]; // Tablica identyfik. watkow

typedef struct {
   int pocz;   // poczatek zakresu
   int kon;    // koniec zakresu
   int numer;  // numer watku
} par_t;

pthread_mutex_t mutex;

void *znajdz(par_t * m);

int main(int argc, char *argv[])
{
  int i,zakres,delta,sum = 0;
  int from, to, nproc,licz,t1,t2;
  par_t  msg;
  if(argc <4) {
     printf("uzycie: pierwsze od do watkow\n");
     exit(0);
  }
  from  = atoi(argv[1]);
  to    = atoi(argv[2]);
  nproc = atoi(argv[3]);
```

```
  if(nproc > NUM_THREADS) nproc = NUM_THREADS;
  printf("zakres od: %d do: %d watkow: %d\n",from,to,nproc);
  zakres = to - from;
  delta = zakres / nproc;
  printf("zakres: %d  delta: %d\n",zakres,delta);
  t1 = time(NULL);
  pthread_mutex_init(&mutex,NULL);

  // Tworzenie watkow -------------
  for (i = 0; i < nproc; i++) {
     pthread_mutex_lock( &mutex );
       msg.pocz = from + delta *i;
       msg.kon  = from + delta * (i+1) -1;
       msg.numer = i;
     pthread_mutex_unlock( &mutex );
     printf("Start: %d pocz: %d kon:
            %d\n",i,msg.pocz,msg.kon);
     pthread_create(&tid[i], NULL, (void *)znajdz,
                    (void *)&msg);
  }

  // Czekanie na zakonczenie ------
  for (i = 0; i < nproc; i++){
       pthread_join(tid[i],(void *)&licz);
       printf("Koniec watku: %d, liczb: %d\n",i,licz);
       sum = sum + licz;
  }
  t2 = time(NULL);
  printf("Watki zak., wynik:%d czas: %d s \n",sum,t2-t1);
  return  0;
}

void *znajdz(par_t* m) {
 int j,x1,x2;
 int lpierw = 0;
 int num ;
 pthread_mutex_lock( &mutex );
   x1 = m->pocz;
   x2 = m->kon;
   num = m->numer;
 pthread_mutex_unlock( &mutex );
 // printf("watek: %d pocz: %d kon: %d \n",num,x1,x2);
 for(j=x1;j<=x2;j++) {
    if(isPrime(j))  lpierw++;
 }
 // printf("Watek: %d, liczb: %d \n",num,lpierw);
 return(( void *) lpierw);
}

int isPrime(int n) {
```

```
   int i;
   for(i=2; i*i <= n; i++) {
      if((n%i) == 0) return 0;
   }
   return 1;
}
```

Przykład 3-8 Znajdowanie liczb pierwszych w przedziale

```
//-----------------------------------------------------------
// Znajdowanie liczb pierwszych  program wielowatkowy
// Ta wersja programu rownowazy obciazenia
// Uruchomienie: ./pierwsze  zakres_od zakres_do watkow
//-----------------------------------------------------------
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 8

pthread_t tid[NUM_THREADS]; // Tablica identyfik. watkow

typedef struct {
   int pocz;    // poczatek zakresu
   int kon;      // koniec zakresu
   int numer;  // numer watku
   int stop;     // zatrzymaj;
} par_t;

void *znajdz(int * num);

pthread_mutex_t mutex;
pthread_cond_t gotowy;
pthread_cond_t start;
par_t msg;
int gotowych = 0; // Ile watkow gotowych do obliczen
int stop = 0;

int main(int argc, char *argv[])
{
  int i,zakres,delta,krok,sum = 0;
  int from, to, nproc,licz,t1,t2,res;
  if(argc <5) {
    printf("uzycie: pierwsze od do watkow krok\n");
    exit(0);
  }
  from  = atoi(argv[1]);
  to    = atoi(argv[2]);
  nproc = atoi(argv[3]);
  krok  = atoi(argv[4]);
  if(nproc > NUM_THREADS) nproc = NUM_THREADS;
  printf("zakres od: %d do: %d watkow: %d krok:
```

```
          %d\n",from,to,nproc,krok);
  zakres = to - from;
  delta = zakres / nproc;
  printf("zakres: %d  delta: %d\n",zakres,delta);
  t1 = time(NULL);
  pthread_mutex_init(&mutex,NULL);
  pthread_cond_init(&gotowy,NULL);
  pthread_cond_init(&start,NULL);

  // Tworzenie watkow -------------
  for (i = 0; i < nproc; i++) {
      res = pthread_create(&tid[i], NULL, znajdz,(void *) i);
      printf("Glowny - utworz watku: %d wynik %d\n",i,res);
  }

  // Obliczenia ------------------------
  for(i=from; i< to; i = i+ krok) {
      printf("Glowny - przedzial od: %d do: %d\n",i,i+krok-1);
      printf("Glowny - czekam\n");
      pthread_mutex_lock( &mutex );
      if(gotowych <= 0) pthread_cond_wait(&gotowy,&mutex);
      printf("Glowny - gotowy watek: %d\n",msg.numer);
      msg.pocz = i;
      msg.kon   = i+krok-1;
      msg. stop = 0
      pthread_mutex_unlock( &mutex );
      pthread_cond_signal(&start);
  }
  printf("Glowny - stop=1\n");
  stop = 1;
  sleep(1);
  pthread_cond_signal(&start);
  pthread_cond_signal(&start);

  // Czekanie na zakonczenie ------
  for (i = 0; i < nproc; i++) {
     pthread_join(tid[i],(void *)&licz);
     printf("Glowny - koniec watku: %d, liczb: %d\n",i,licz);
     sum = sum + licz;
  }
  t2 = time(NULL);
  printf("Watki zak., wynik:%d czas: %d s \n",sum,t2-t1);
  return  0;
}

void *znajdz(int * num) {
 int j,x1,x2;
 int lpierw = 0;
 printf("Start watku: %d \n",num);
 do {
```

```
   pthread_mutex_lock( &mutex );

   gotowych++;
   printf("Watek: %d wysyla gotowosc\n");
   pthread_cond_signal(&gotowy);
   printf("Watek: %d czeka na start\n");
   pthread_cond_wait(&start,&mutex);
   x1 = msg.pocz;
   x2 = msg.kon;
   pthread_mutex_unlock( &mutex );
   printf("Watek: %d, zakres od %d do %d \n",num,x1,x2);
   if(stop == 1) break;
   for(j=x1;j<=x2;j++) {
      if(isPrime(j))  lpierw++;
   }
   printf("Watek: %d, od:%d do:%d liczb: %d
          \n",num,x1,x2,lpierw);
  } while(stop != 1);
  printf("Watek: %d koniec\n",num);
  return(( void *) lpierw);
}

int isPrime(int n) {
   int i;
   for(i=2; i*i <= n; i++) {
      if((n%i) == 0) return 0;
   }
   return 1;
}
```

Przykład 3-9 Szukanie liczb pierwszych – wersja równoważąca obciążenia

4 Blokady czytelników i pisarzy

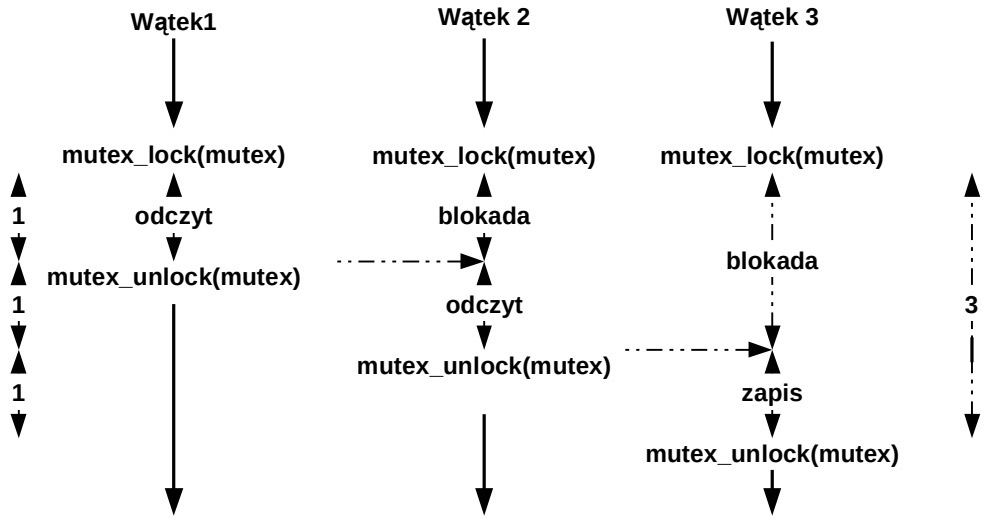
Blokady typu wzajemne wykluczanie są zbyt restrykcyjne co prowadzi do ograniczenia równoległości i spadku efektywności aplikacji.
Można je osłabić wiedząc czy przeprowadzana jest operacja odczytu czy zapisu. Implementują to blokady czytelników i pisarzy (ang. *Readers writers locks*).

Zasada działania blokad czytelników i pisarzy:

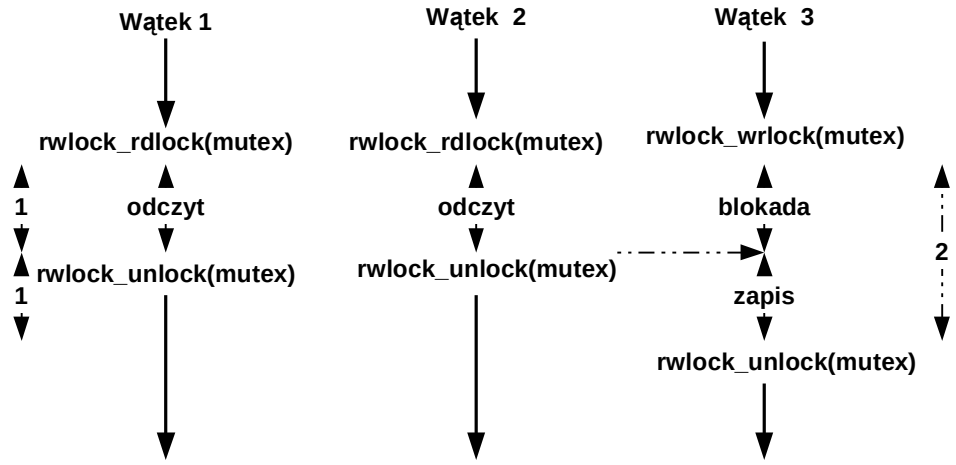
- Odczyt może być wykonywany współbieżnie do innych odczytów
- Zapis musi być wykonywany w trybie wyłącznym względem innych zapisów lub odczytów.

Stan blokady:

- Wolna
- Zajęta do odczytu być może przez wiele wątków czytających
- Zajęta do zapisu



Rys. 4-1 Dwa wątki uczytające i jeden piszący używają wspólnego zasobu chronionego przez mutex. Czas wykonania 3 jednostki.



Rys. 4-2 Dwa wątki uczytające i jeden piszący używają wspólnego zasobu chronionego przez blokady czytelników i pisarzy. Odczyty mogą być prowadzone współbieżnie. Czas wykonania 2 jednostki.

Inicjacja blokady

```
int pthread_rwlock_init(pthread_rwlock_t * rwlock,
 pthread_rwlockattr_t * attr)
```

rwlock	Zadeklarowana i zainicjowana zmienna typu **pthread_rwlock_t**
attr	Atrybuty blokady lub NULL gdy maja być domyslne

Zajęcie blokady do odczytu

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
```

Wątek wykonujący funkcję blokuje się gdy blokada jest zajęta do zapisu. Gdy nie zajmuje blokadę do odczytu gdy nie została już wcześniej zajęta do odczytu.

Zajęcie blokady do zapisu

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

Wątek wykonujący funkcję blokuje się gdy blokada jest zajęta do zapisu lub odczytu. Gdy nie zajmuje blokadę do zapisu.

Zwolnienie blokady

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock)

Funkcja zdejmuje blokadę nałożoną jako ostatnią przez bieżący wątek. Jeżeli istnieją inne blokady założone na obiekt to pozostają. Jeżeli jest to ostatnia blokada i istnieją wątki czekające na jej zwolnienie to jeden z nich zostanie odblokowany. Który zależy to od implementacji.

Nieblokujace zajęcie blokady do zapisu

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock)
Gdy blokada jest wolna nastepuje jej zajęcie do zapisu. Gdy jest zajeta funkcja nie blokuje wątku biezącego i zwraca kod błędu.

Czasowo ograniczone zajęcie blokady do zapisu

int pthread_rwlock_timedwrlock(pthread_rwlock_t * ***rwlock*****, struct timespec *** ***abs_timeout***);

Jeżeli po upływie zadanego czasu *abs_timeout* blokada nie zostanie zdjęta funkcja odblokuje się zwracając kod błędu.

Nieblokujace zajęcie blokady do odczytu

int pthread_rwlock_tryrdlock(pthread_rwlock_t ***rwlock**)

Gdy blokada jest wolna lub zajeta do odczytu nastepuje jej zajęcie do odczytu. Gdy jest zajeta funkcja nie blokuje wątku biezącego i zwraca kod błędu.

Czasowo ograniczone zajęcie blokady do odczytu

int pthread_rwlock_timedrdlock(pthread_rwlock_t * ***rwlock*****, struct timespec *** ***abs_timeout***);

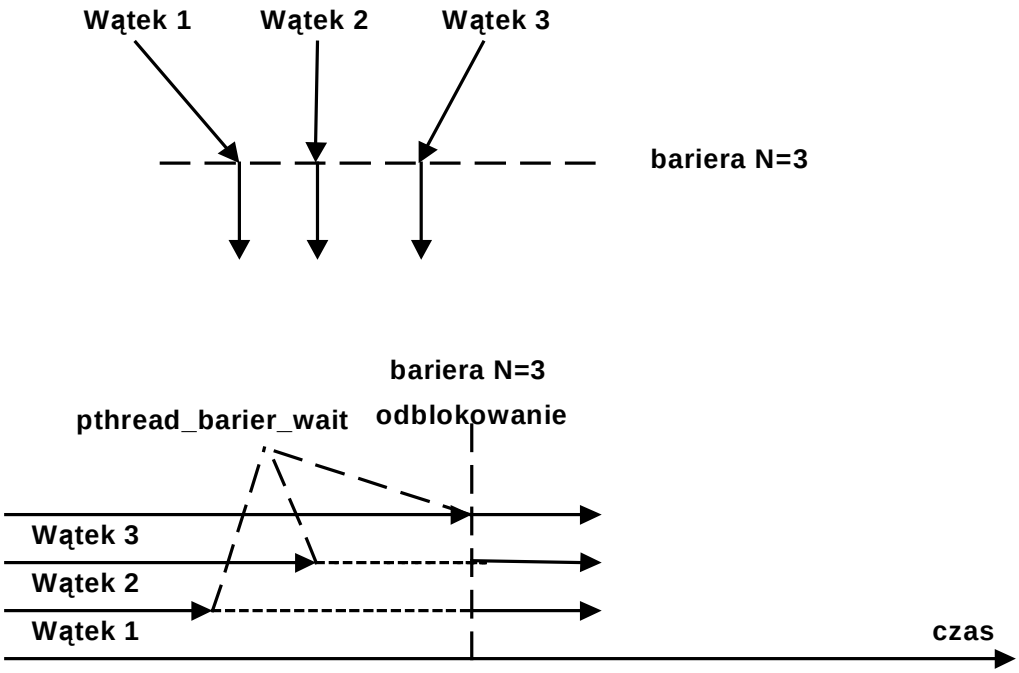
Jeżeli po upływie zadanego czasu *abs_timeout* blokada nie zostanie zdjęta funkcja odblokuje się zwracając kod błędu.

Skasowanie blokady

int pthread_rwlock_destroy(pthread_rwlock_t * ***rwlock*****)**

5 Bariery

Bariera jest narzędziem do synchronizacji procesów działających w ramach grup. Wywołanie funkcji **`pthread barrier wait(...)`** powoduje zablokowanie zadania bieżącego do chwili gdy zadana liczba watków zadań nie wywoła tej procedury.



Rys. 5-1 Działanie bariery

Inicjacja bariery

```
int pthread_barrier_init( pthread_barrier_t * barrier,
pthread_barrierattr_t * attr
unsigned int count )
```

`barrier`	Zadeklarowana zmienna typu **`pthread_barierr_t.`**
`attr`	Atrybuty. Gdy NULL użyte są atrybuty domyślne
`count`	Licznik bariery

Funkcja powoduje zainicjowanie bariery z wartością licznika count.

Czekanie na barierze

```
int pthread_barrier_wait( pthread barrier t * barrier )
```

barrier Zadeklarowana i zainicjowana zmienna typu **pthread_barierr_t.**

Funkcja powoduje zablokowanie wątku bieżącego na barierze. Gdy count wątków zablokuje się na barierze to wszystkie zostaną odblokowane.
Funkcja zwraca **BARRIER SERIAL THREAD** dla jednego z wątków (wybranego arbitralnie) i 0 dla wszystkich pozostałych wątków. Wartość licznika będzie taka jak przy ostatniej inicjacji bariery.

Kasowanie bariery

```
int pthread barrier destroy( pthread barrier t * barrier )
```

barrier Zadeklarowana i zainicjowana zmienna typu **pthread_barierr_t.**

Funkcja powoduje skasowanie bariery.

```
#include <sys/types.h>
#include <pthread.h>
#include <malloc.h>

pthread_barrier_t * my_barrier;

void * thread1(void * arg){
    printf("Watek 1 przed bariera\n");
    pthread_barrier_wait(my_barrier);
    printf("Watek 1 po barierze \n");
}

void * thread2(void * arg){
   printf("Watek 2 przed bariera\n");
   pthread_barrier_wait(my_barrier);
   printf("Watek 2 po barierze \n");
}


int main(){
   pthread_t w1,w2;
   my_barrier =
(pthread_barrier_t*)malloc(sizeof(pthread_barrier_t));
   pthread_barrier_init(my_barrier, NULL, 2);
   pthread_create(&w1, 0, thread1, 0);
   pthread_create(&w2, 0, thread2, 0);
   pthread_join(w1, 0);
   pthread_join(w2, 0);
   return 0;
}
```

Przykład 5-1 Przykład użycia bariery

6 Wirujące blokady

Wirujące blokady są środkiem zabezpieczania sekcji krytycznej. Wykorzystują jednak czekanie aktywne zamiast przełączenia kontekstu wątku tak jak się to dzieje w muteksach.

Inicjacja wirującej blokady

```
int pthread spin init( pthread spinlock t *blokada,
int pshared )
```

blokada Identyfikator wirującej blokady **pthread spinlock t**

pshared
- **PTHREAD PROCESS SHARED –** na blokadzie mogą operować wątki należące do różnych procesów
- **PTHREAD PROCESS PRIVATE –** na blokadzie mogą operować tylko wątki należące do tego samego procesu

Funkcja inicjuje zasoby potrzebne wirującej blokadzie. Każdy proces, który może sięgnąć do zmiennej identyfikującej blokadę może jej używać. Blokada może być w dwóch stanach:
- Wolna
- Zajęta

Zajęcie blokady

```
int pthread_spin_lock( pthread spinlock t * blokada)
```

blokada Identyfikator wirującej blokady – zmienna typu **pthread_spinlock_t**

Działanie funkcji zależy od stanu blokady. Gdy blokada jest wolna następuje jej zajęcie. Gdy blokada jest zajęta wątek wykonujący funkcję **pthread_spin_lock(...)** ulega zablokowaniu do czasu gdy inny wątek nie zwolni blokady wykonując funkcję **pthread_spin_unlock(...)**.

Próba zajęcia blokady

int pthread_spin_trylock(pthread spinlock t * blokada)

blokada Identyfikator wirującej blokady – zmienna typu **pthread_spinlock_t**

Działanie funkcji zależy od stanu blokady. Gdy blokada jest wolna następuje jej zajęcie – funkcja zwraca wartość **EOK.** Gdy blokada jest zajęta następuje natychmiastowy powrót i funkcja zwraca stałą **EBUSY**.

Zwolnienie blokady

int pthread_spin_unlock(pthread spinlock t * blokada)

blokada Identyfikator wirującej blokady – zmienna typu **pthread_spinlock_t**

Działanie funkcji zależy od stanu blokady. Gdy są wątki czekające na zajęcie blokady to jeden z nich zajmie blokadę. Gdy żaden wątek nie czeka na zajęcie blokady będzie ona zwolniona.

Skasowanie blokady

int pthread_spin_destroy(pthread spinlock t * blokada)

blokada Identyfikator wirującej blokady – zmienna typu **pthread_spinlock_t**

Funkcja zwalnia blokadę i zajmowane przez nią zasoby.

7 Wątki w środowisku wieloprocesorowym

W systemie Linux począwszy od wersji 2.0 obsługiwana jest architektura wieloprzetwarzania symetrycznego (ang. *Symmetric Multi Processing*) SMP. W modelu tym zakłada się że wszystkie procesory mają takie same właściwości.

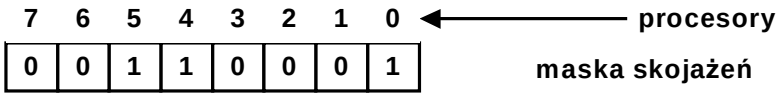
Maska skojarzeń

Być może w pewnych aplikacjach korzystne jest wskazanie mechanizmowi szeregowania na jakich procesorach mają się pewne wątki wykonywać. Mechanizm ten nosi nazwę maski skojarzeń (ang. *affinity mask*).

Maska skojarzeń – mapa bitowa gdzie każdy bit odpowiada pojedynczemu procesorowi. Maska skojarzeń utrzymywana jest dla każdego wątku.

- Gdy bit i zawiera 1 – watek może się wykonywać na procesorze i.
- Gdy bit i zawiera 0 – nie watek może się wykonywać na procesorze i.

Zbiór procesorów na którym może się wykonywać dany wątek jest iloczynem bitowym maski skojarzeń i maski odpowiadającej procesorom rzeczywiście zainstalowanym w systemie.



Rys. 7-1 Przykład maski skojarzeń - wątek może się wykonywać na procesorze 0, 4, 5

Ustawianie maski skojarzeń:

```
int pthread_setaffinity_np(pthread_t thread, size_t
cpusetsize,cpu_set_t *cpuset);
```

Testowanie maski skojarzeń

```
int pthread_getaffinity_np(pthread_t thread, size_t
cpusetsize,cpu_set_t *cpuset);
```

gdzie:

thread	Identyfikator wątku
cpusetsize	Długość maski skojarzeń w bajtach
cpuset	Maska skojarzeń

Typ **cpu_set_t** – maska bitowa długości do 1024

void CPU_ZERO(cpu_set_t *set)	Zerowanie maski
void CPU_SET(int cpu,cpu_set_t *set)	Ustawianie bitu cpu
void CPU_CLR(int cpu,cpu_set_t *set)	Zerowanie bitu cpu
int CPU_ISSET(int cpu,cpu_set_t *set)	Testowanie bitu cpu
int CPU_COUNT(cpu_set_t *set)	Liczenie ustawionych bitów

Tabela 7-1 Funkcje operujące na maskach skojarzeń

Zastosowanie maski skojarzeń ma sens wtedy gdy skierujemy intensywnie komunikujące się wątki na jeden procesor. Spowoduje to ograniczenie liczby chybień w pamięci podręcznej i do przyspieszenia działania.

Wątkom można przypisać procesory jeszcze przed ich uruchomieniem ustawiając odpowiednio ich atrybuty za pomocą funkcji.

```
int pthread_attr_setaffinity_np(pthread_attr_t *attr,
    size_t cpusetsize, const cpu_set_t *cpuset)
```

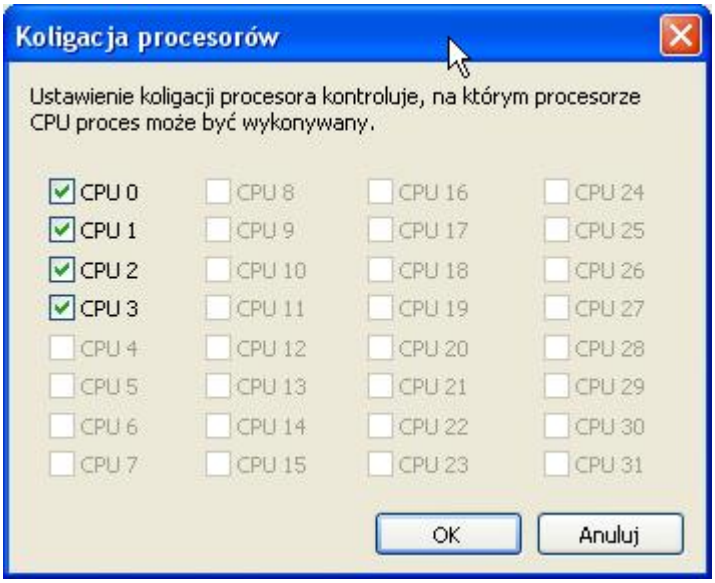
```
int pthread_attr_getaffinity_np(pthread_attr_t *attr,
    size_t cpusetsize, cpu_set_t *cpuset)
```

Znaczenie argumentów takie jak poprzednio.

```
int main(int argc, char *argv[]) {
  int s, j;
  cpu_set_t cpuset;
  pthread_t thread;
  thread = pthread_self();
  /* Ustaw maske skojarzeń włączając CPU 0 do 7 */
  CPU_ZERO(&cpuset);
  for (j = 0; j < 8; j++) CPU_SET(j, &cpuset);
  s = thread_setaffinity_np(thread,sizeof(cpu_set_t),
      &cpuset);

  /* Testuj maske skojarzeń */
  s = thread_getaffinity_np(thread,sizeof(cpu_set_t),
      &cpuset);
  printf("Wynik:\n");
   for (j = 0; j < CPU_SETSIZE; j++)
        if (CPU_ISSET(j, &cpuset))
           printf("    CPU %d\n", j);
   exit(EXIT_SUCCESS);
 }
```

Przykład 7-1 Ustawianie i testowanie maski skojarzeń – dozwolone procesory 0 do 7



Rys. 7-2 Koligacja procesorów w Windows