

# Wątki

1.	Własności wątków.....	2
1.1.	Informacje wstępne.....	2
1.2.	Atrybuty i zasoby wątków .....	6
1.3.	Wątki i procesy – porównanie.....	9
2.	Biblioteka pthreads .....	10
2.1.	Tworzenie i kończenie wątków .....	11
2.2.	Atrybuty wątku .....	15
2.3.	Przekazywanie danych do wątku i pobieranie wyników.....	19
2.4.	Dostęp do wspólnych danych.....	23
2.5.	Zmienne warunkowe.....	29
3.	Blokady czytelników i pisarzy .....	40
4.	Bariery .....	43
5.	Wirujące blokady.....	46
6.	Wątki w środowisku wieloprocesorowym .....	48
7.	Literatura: .....	51

## 1. Własności wątków

### 1.1. Informacje wstępne

Współczesne systemy operacyjne oferują użytkownikowi następujące ważne abstrakcje:

- Plik - ujednolicone urządzenia zewnętrzne w tym system pamięci trwałej
- Pamięć wirtualna – proces ma wrażenie indywidualnej pamięci
- Proces a w jego ramach wątek – proces otrzymuje wiele zwirtualizowanych procesorów

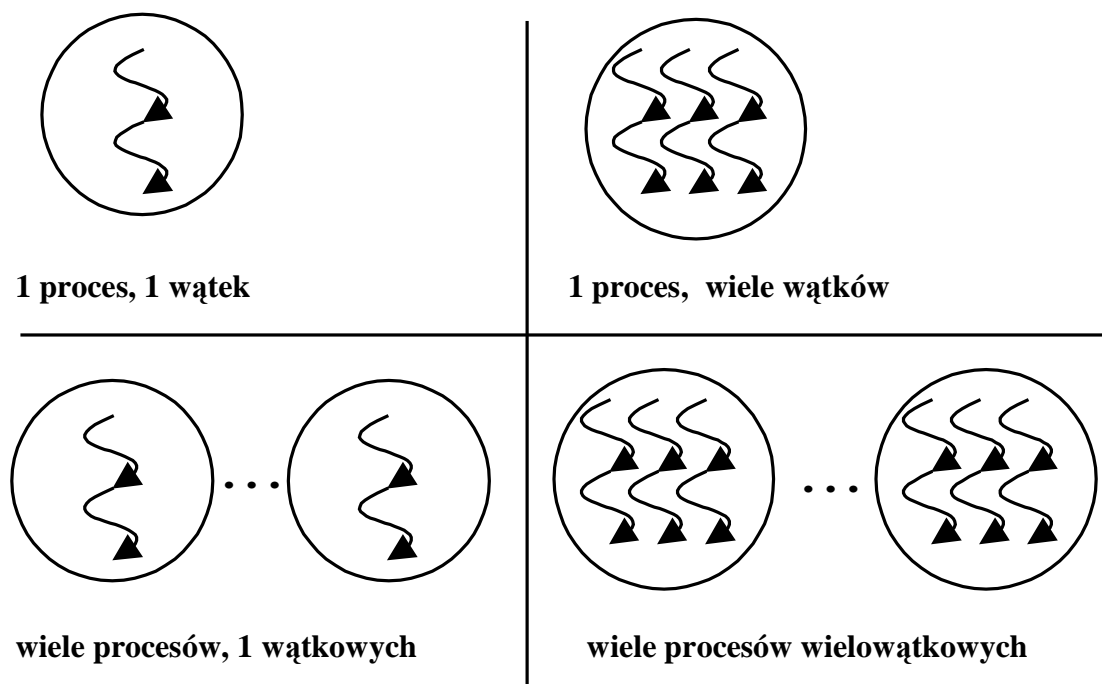
Tradycyjna implementacja procesu ma jeden wątek sterowania. W nowszych systemach pojęcie procesu zostało rozszerzone – dopuszcza się istnienie wielu wątków sterowania. Proces można podzielić na zbiór wątków i zestaw zasobów.

**Proces** – pojemnik na zasoby w ramach którego wykonują się wątki.

**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

Wiele aplikacji ma do wykonania czynności które mogą być wykonane współbieżnie, ale muszą dzielić dane i inne zasoby. Można tu zastosować procesy lub wątki.



Rys. 1-1 Podstawowe modele przetwarzania

Procesy – duża izolacja wzajemna, komunikacja wymaga specjalnych mechanizmów, duży stopień bezpieczeństwa.

Wątki – mała wzajemna izolacja, prosta komunikacja między wątkami, mniejszy stopień bezpieczeństwa.

Co daje zastosowanie modelu wątków (według [1]) :

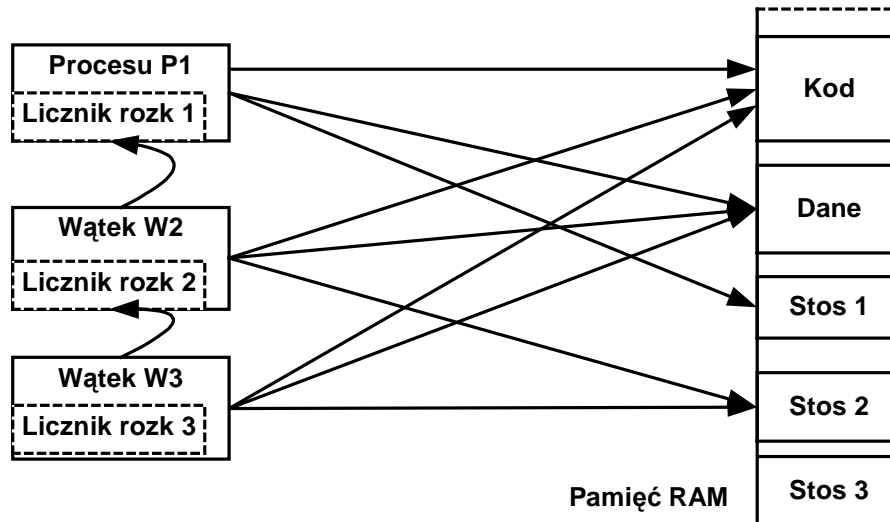
- Abstrakcja programistyczna – ułatwiają podział dużego problemu na części
- Wykorzystanie procesorów wielordzeniowych – rzeczywista równoległość
- Poprawa czasu reakcji
- Szybsze przełączanie kontekstu
- Oszczędność pamięci (w porównaniu z zastosowaniem analogicznej liczby procesów)

Inne opinie:

*„Wątki są dla osób które nie potrafią zaprogramować automatu stanu.”*  
(Alan Cox)

### Zasada działania

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



Rys. 1-2 Wątki 2 i 3 wykonywane w ramach procesu P1

Każdy z wątków ma niezależny:

- Stos – przechowuje zmienne lokalne wątku
- Licznik rozkazów – określa indywidualną ścieżką
- Zestaw indywidualnych atrybutów.

Indywidualne atrybuty wątku przechowywane są w obszarach TLS (ang. *Thread Local Storage*).

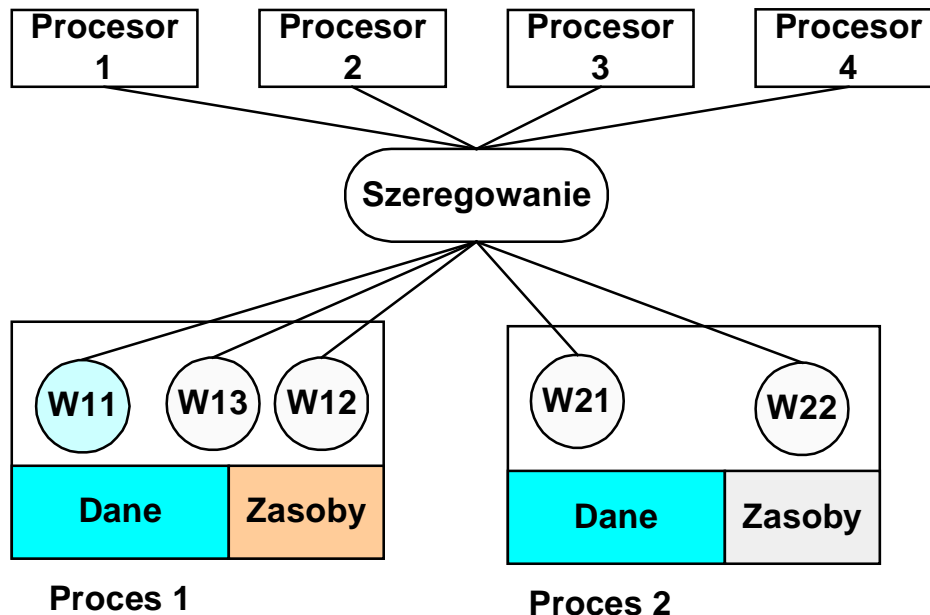
### Własności wątków

- 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.

## Wątki

## Rzeczywista równoległość

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



Rys. 1-3 Procesy wielowątkowe wykonywane na wielu procesorach

W systemie Linux występują dwie biblioteki dotyczące wątków:

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

Aby zorientować się z którą mamy do czynienia należy użyć polecenia:

```
getconf GNU_LIBPTHREAD_VERSION
```

```
root@maria:/home/juka/prog/beagle# getconf GNU_LIBPTHREAD_VERSION
NPTL 2.11.3
```

## Funkcja clone

W Linuksie tworzenia wątków odbywa się za pomocą funkcja clone. Pozwala ona na dzielenie z utworzonym procesem segmentu kodu i innych obiektów.

```
int clone(int (*fn)(void *), void *child_stack,
         int flags, void *arg, ...
         pid_t *ptid, struct user_desc *tls,
         pid_t *ctid );
```

<b>fn</b>	Funkcja według której tworzony jest kod
<b>child_stack</b>	Wskaźnik na położenie stosu
<b>flags</b>	Flagi
<b>arg</b>	Argumenty przekazywane do funkcji tworzonego procesu
<b>ptid</b>	Identyfikator nowego procesu
<b>tls</b>	Thread local storage

Funkcja clone zazwyczaj nie jest wywoływana jawnie ale za pomocą bibliotek pthreads – funkcja pthreads\_create(...)

## 1.2. Atrybuty i zasoby wątków

Wątek dzieli ze swym procesem macierzystym następujące zasoby:

- Dane statyczne (segment danych, sterty)
- Identyfikator procesu PID i procesu macierzystego PPID
- Grupa procesów PGRP i identyfikator sesji SID
- Terminal sterujący
- Identyfikator użytkownika UID, EUID i grupy GID, EGID
- Deskryptory otwartych plików, blokady plików
- Maskę tworzenia plików (umask)
- Środowisko
- Katalog macierzysty (ang. *home directory*) i główny (ang. *root directory*)
- Funkcje obsługi sygnałów
- Wartość parametru nice
- Limity zasobów (setrlimit)
- Timery
- Liczniki dotyczące czasu użycia procesora

---

## Wątki

### Wątki posiadają następujące atrybuty i zasoby:

1. Identyfikator wątku PID, TID (*ang. Thread Identifier*) – każdy wątek ma unikalny w identyfikator. Jest to liczba całkowita. W Linuxie każdy wątek ma inny PID
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. Maskę 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 początek stosu, kod ewentualnego błędu `errno`, funkcja czyszcząca i inne dane.
6. Procedura zakończenia (*ang. Cancellation Handler*) - gdy wątek się kończy wykonywana jest procedura zakończenia w ramach której zwalniane są zasoby wątku.

Dla Linuxa także:

- Przydział wątków do procesora (*ang. CPU affinity*)
- Zakresy kontroli w trybie User (*ang. Capabilities*)

## Jak wyświetlić wykonywane wątki?

```
ps -ef
UID          PID  PPID  C  STIME TTY          TIME CMD
root           1     0  0 Nov24 ?           00:00:04 init [2]
...
root        1147     1  0 Nov24 ?           00:00:00 /usr/sbin/rsyslogd -c4
```

Przykład 1-1 Proces rsyslogd – wątki nie są widoczne

```
ps -eLf
UID      PID  PPID  LWP  C  NLWP  STIME TTY          TIME CMD
...
root    1147     1  1147  0    3 Nov24 ?           00:00:00 /usr/sbin/rsyslogd -c4
root    1147     1  1177  0    3 Nov24 ?           00:00:00 /usr/sbin/rsyslogd -c4
root    1147     1  1178  0    3 Nov24 ?           00:00:00 /usr/sbin/rsyslogd -c4
```

Przykład 1-2 Wątki procesu rsyslogd

### Modele wątkowości:

- Wątki poziomu jądra – wątki tworzone i administrowane przez jądro
- Wątki na poziomie użytkownika – w ramach jednego procesu użytkownik tworzy wiele wątków przełączanych wewnątrz procesu
- Model mieszany

### Inne rozwiązania:

- Współprogramy (ang. *coroutines*)
- Włókna (ang. *fibers*)

Podobne do podprogramów, zapewniają jednostkę wykonawczą jeszcze lżejszą niż wątek.



### 1.3. Wątki i procesy – porównanie

#### 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 wieloprocessorowego SMP w systemach ze wspólną pamięcią (*ang. Symetrical Multi Procesing*).

#### Wady modelu wielowątkowego:

- Mniejsze bezpieczeństwo aplikacji

#### Zagadnienia:

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. 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).

### 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
Funkcje SMP	Sterowanie wykonywaniem wątków w środowisku wieloprocessorowym

## 2.1. Tworzenie i kończenie wątków

<code>pthread_create</code>	Tworzenie wątku
<code>pthread_exit</code>	Zakończenie wątku bieżącego
<code>pthread_join</code>	Czekanie na zakończenie wątku
<code>pthread_attr_init</code>	Inicjacja atrybutów wątku
<code>pthread_self</code>	Pobranie identyfikatora wątku
<code>pthread_yield</code>	Zwolnienie procesora
<code>pthread_cancel</code>	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)
```

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

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

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

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

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

Przykład 2-1 Tworzenie wątku za pomocą funkcji `pthread_create`

```
$/thread1
MWMWMWMWMWMWMWMWMWMWMWMWMWMWMWM
```

Wynik 2-1 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 funkcji:

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

**status**      Kod powrotu wątku (będący wskaźnikiem) przekazywany jako status do funkcji:  
**pthread\_join(pthread\_t \*thread, void \*status)**

Uwaga!

Zmienna wskazywanej w funkcji `pthread_exit` nie może być umieszczona w pamięci lokalnej wątku gdyż po jego zakończeniu obszar ten jest zwalniany i wartość zmiennej jest nieokreślona.

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 (wykonującej 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)
```

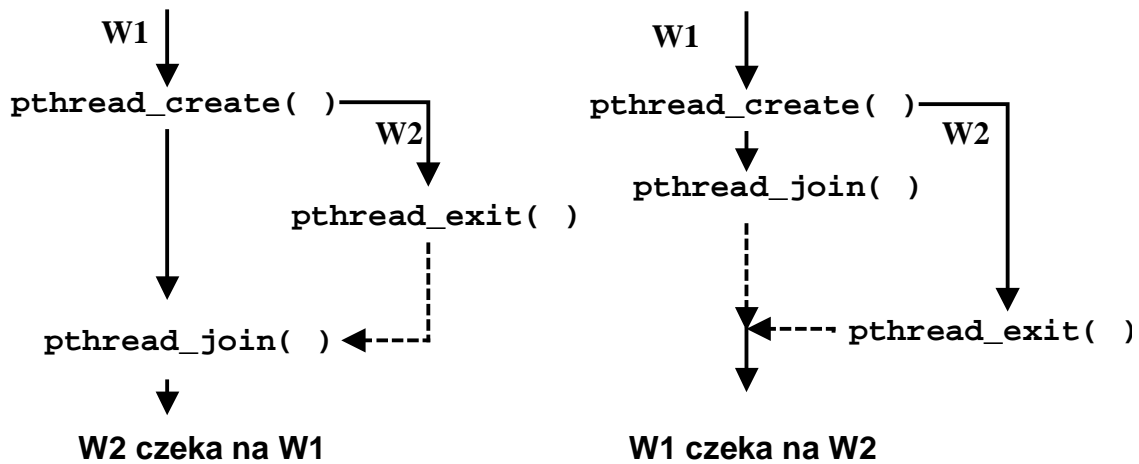
**thread**      identyfikator wątku – wartość nadawana przez funkcję  
**status**      Kod powrotu zwracany przez zakończony wątek

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

---

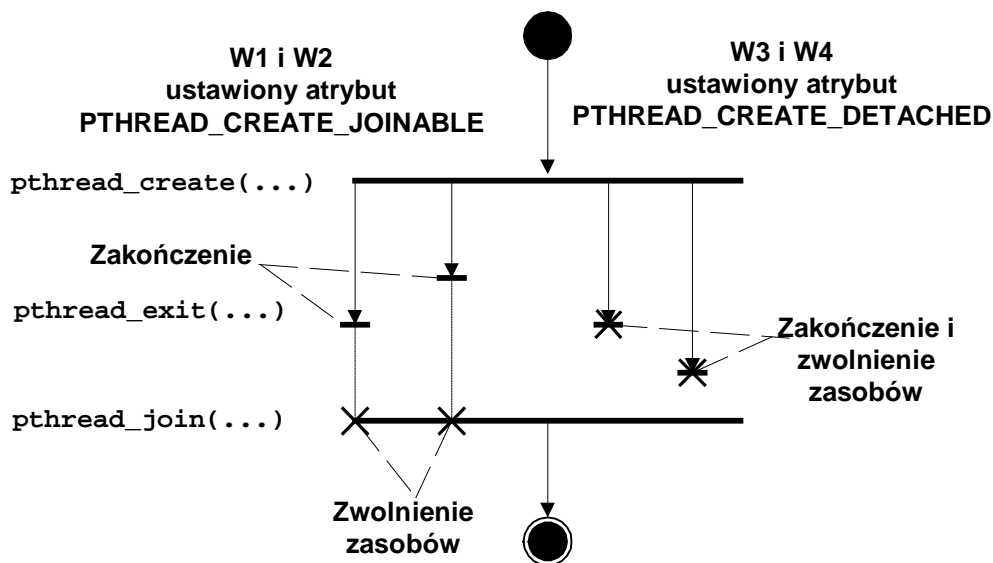
## **Wątki**

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



Rys. 2-1 Zakończenia wątków

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



Rys. 2-2 Wątki dołączalne i nie dołączalne

## 2.2. Atrybuty wątku

Atrybuty wątku są to jego dodatkowe własności jak:

- dołączalność,
- priorytet,
- strategia szeregowania,
- strategia unikania inwersji priorytetów (w Linuksie brak),
- 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:

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`.
3. Parametry szeregowania – różne informacje (między innymi priorytet) używane 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	<code>PTHREAD_CREATE_JOINABLE</code>
Strategia szeregowania	<code>PTHREAD_INHERIT_SCHED</code>
Parametry szeregowania	Dziedziczone z procesu macierzystego
Rozmiar stosu	4 KB
Adres stosu	<code>NULL</code>

Tabela 2-1 Atrybuty wątku i ich wartości domyślne

Atrybut	Funkcja testowania	Funkcja ustawiania
Dołączalność	<code>attr_getdetachstate()</code>	<code>attr_setdetachstate()</code>
Strategia szeregowania	<code>attr_getschedpolicy()</code>	<code>attr_setschedpolicy()</code>
Parametry szeregowania	<code>attr_getschedparam()</code>	<code>attr_setschedparam()</code>
Rozmiar stosu	<code>attr_getstacksize()</code> , <code>attr_getstacklazy()</code>	<code>attr_setstacksize()</code> , <code>attr_setstacklazy()</code>
Adres stosu	<code>attr_getstackaddr()</code>	<code>attr_setstackaddr()</code>

Tabela 2-2 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 2-2 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("warki 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 2-3 Tworzenie, wykonanie i kończenie wątków

## 2.3. 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:

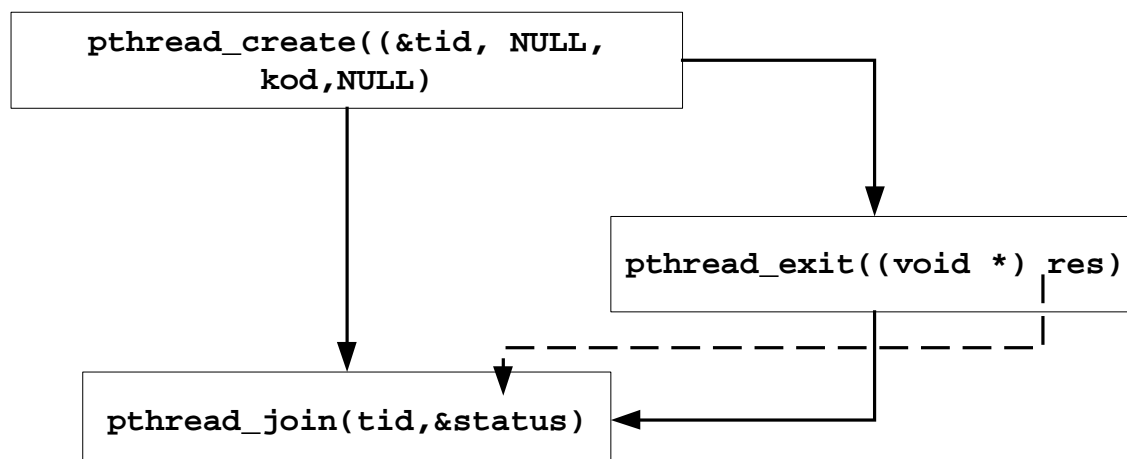
- Wykonanie `return res`
- Wykonanie funkcji `pthread_exit(void *res)`

Wartość ta może być odczytana w funkcji:

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

Uwaga!

Zmienna wskazywana w funkcji `pthread_exit` nie może być umieszczona w pamięci lokalnej wątku gdyż po jego zakończeniu obszar ten jest zwalniany i wartość zmiennej jest nieokreślona.



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

int main(int argc, char *argv[]){
    param_t arg;
    int wynik,res[NUM_THREADS];
    ...
    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;
    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[num] = ...;
    return((void *) res[num]);
}
```

Przykład 2-4 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.

## 2.4. 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.

<code>pthread_mutex_init</code>	Inicjacja muteksu
<code>pthread_mutex_lock</code>	Zajęcie muteksu
<code>pthread_mutex_trylock</code>	Zajęcie muteksu gdy wolny
<code>pthread_mutex_timedlock</code>	Zajęcie muteksu – czekanie ograniczone czasowo
<code>pthread_mutex_unlock</code>	Zwolnienie muteksu
<code>pthread_mutex_destroy</code>	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 obiektu typu muteks musi być zadeklarowany. Przykładowo muteks o nazwie `blokada` deklaruje się jak poniżej.

```
pthread_mutex_t blokada;
```

### Inicjacja muteksu

Przed użyciem muteksu 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.

---

## Wątki

Zainicjowany muteks pozostaje w stanie odblokowania.

Implementacja LinuxThreads uwzględnia tylko jeden atrybut muteksu: typ muteksu który może być: **normal**, **recursive**, **error-checking**.

Typ muteksu określa zachowanie wątku który posiada muteks i próbuje go powtórnie zablokować. Domyślny typ muteksu to **normal**.

PTHREAD_MUTEX_NORMAL	Gdy wątek posiada muteks i próbuje go ponownie zająć następuje zakleszczenie
PTHREAD_MUTEX_ERRORCHECK	Gdy wątek posiada muteks i próbuje go ponownie zająć funkcja zajęcia muteksu kończy się błędem
PTHREAD_MUTEX_RECURSIVE	Utrzymywany jest licznik zajętości, zajęcie zwiększa licznik, zwolnienie zmniejsza. Gdy osiągnie zero wątek się odblokuje.

Tab. 2-1 Typy muteksów

### 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.

### Zablokowanie dostępu do zasobu gdy wolny

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

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

Działanie funkcji zależy od stanu w jakim znajduje się muteks.

---

## Wątki



1. Gdy muteks jest wolny, następuje jego zablokowanie.
2. Gdy muteks jest zajęty, brak blokady, funkcja zwraca -1

### 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 wątek się blokuje ale po upływie danego okresu ulega odblokowaniu.

### Odblokowanie dostępu do zasobu

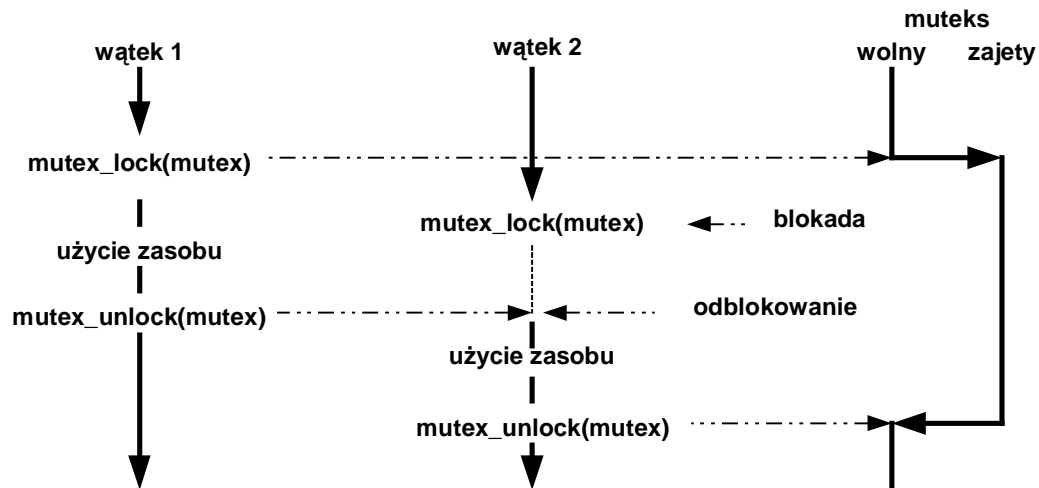
Użyty i zablokowany wcześniej zasób powinien być zwolniony. Zwolnienie zasobu odbywa się poprzez wywołanie funkcji:

```
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. 2-3 Dwa wątki używają wspólnego zasobu chronionego przez mutex.

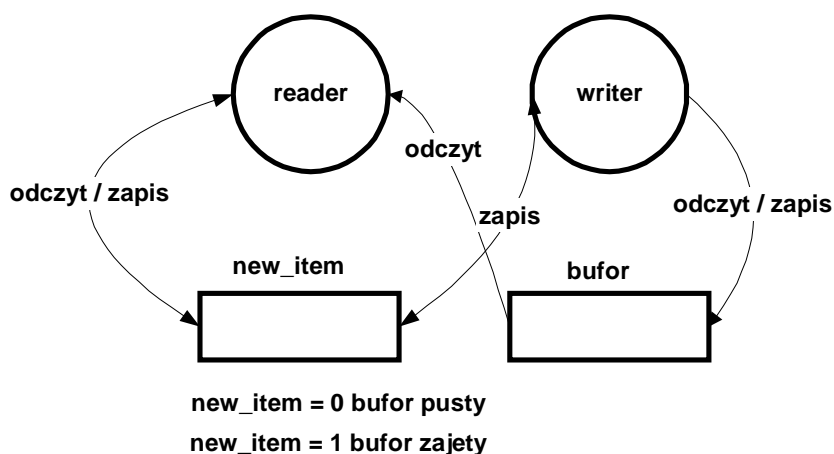
### Kasowanie muteksu

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 błędu.

Przykład: odczyt – zapis z bufora



Rys. 2-1 Odczyt – zapis z bufora

```
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);
}
```

---

## Wątki

```
// 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 2-5 Problem producenta / konsumenta – rozwiązanie z odpytywaniem

### **futex**

futex - mechanizm synchronizacji niższego poziomu (ang. *Fast User Space Mutexes*)

Mechanizm synchronizacji usytuowany w przestrzeni użytkownika, stosowany do tworzenia mechanizmów wyższego poziomu. Jest on specyficzny dla Linuxa, nie należy do standardu POSIX.

## 2.5. 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 `pthread_cond_t`.

Najważniejsze operacje związane ze zmiennymi warunkowymi są dane poniżej.

<code>pthread_cond_init</code>	Inicjacja zmiennej warunkowej.
<code>pthread_cond_wait</code>	Czekanie na zmiennej warunkowej
<code>pthread_cond_timedwait</code>	Ograniczone czasowo czekanie na zmiennej warunkowej
<code>pthread_cond_signal</code>	Wznowienie wątku zawieszonoego w kolejce danej zmiennej warunkowej.
<code>pthread_cond_broadcast</code>	Wznowienie wszystkich wątków zawieszonych w kolejce danej zmiennej warunkowej.
<code>pthread_cond_destroy</code>	Skasowanie zmiennej warunkowej z zwolnieniem jej zasobów.

### Inicjacja zmiennej warunkowej

```
int pthread_cond_init(pthread_cond_t *zw,  
pthread_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(pthread_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 `pthread_cond_signal(&zw)` zablokowany wątek zostanie odblokowany a blokada `mutex` zwolniona.

### Zawieszenie wątku w oczekiwaniu na sygnalizację z ograniczeniem czasowym

```
int pthread_cond_timedwait(pthread_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 `pthread_cond_signal(&zw)` zablokowany wątek zostanie odblokowany a blokada mutex zwolniona. Gdy przekroczony zostanie czas oczekiwania watek będzie odblokowany.

### **Wznowienie zawieszzonego wątku**

```
int pthread_cond_signal(pthread_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_broadcast(pthread_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:

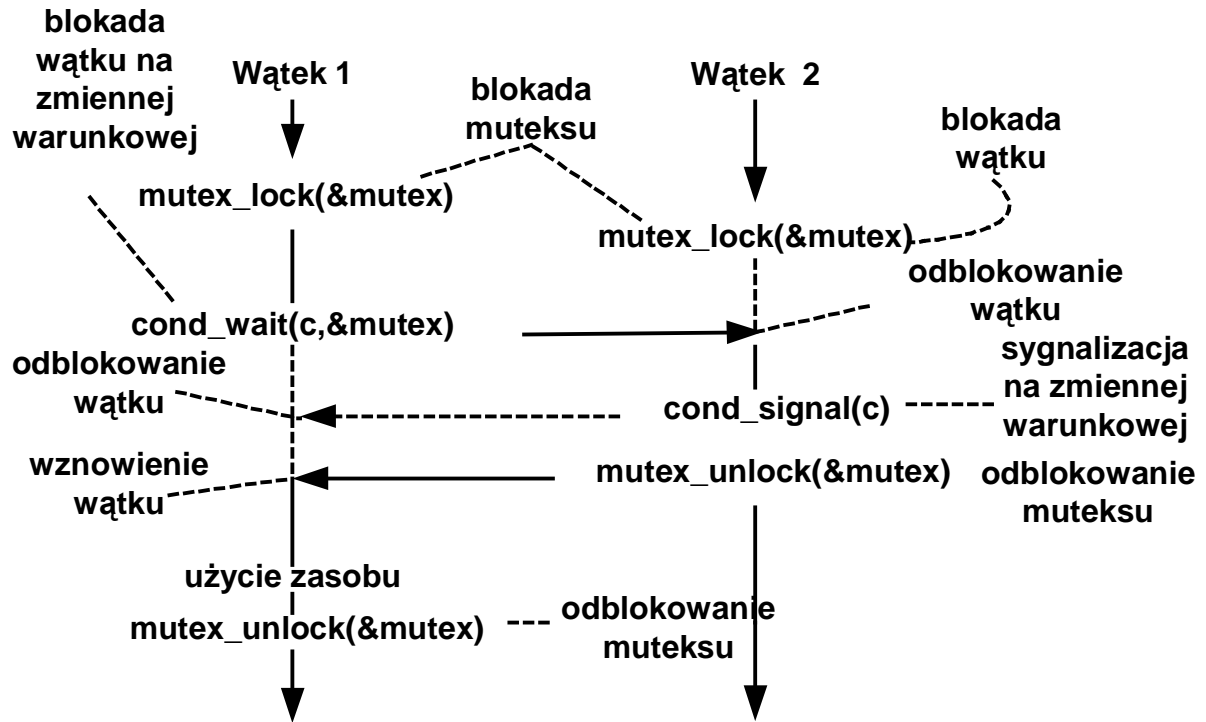
```
pthread_mutex_lock(&m)
...
while( ! warunek )
    pthread_cond_wait( &cond, &m)
...
pthread_mutex_unlock(&m)
```

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. 2-4 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 czytelnii, -1 gdy jest tam pisarz
int count;

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

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

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

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

---

## Wątki

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

Przykład 2-6 Rozwiązanie problemu czytelników i pisarzy

## Rozwiązanie problemu producenta - konsumenta

```
#include <stdio.h>
#include <pthread.h>
#define N 4
int head ,tail,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[tail] = cnt++;
        tail = (tail+1) %N;
        licznik++;
        pthread_cond_signal(&pelne);
        pthread_mutex_unlock( &mutex );
        printf("Prod%d tail: %d wst: %d\n", num, tail,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[head];
        head = (head+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 );
    }
}
```

## Wątki

```
int main( void ) {
    int head = 0;
    int tail = 0;
    int licznik = 0;
    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 2-7 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
int wynik[NUM_THREADS];     // Tablica z wynikami

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;
```

## Wątki

```
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;
}
```

## Wątki

```
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);
    wynik[num] = lpierw;
    return(&wynik[num]);
}

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

Przykład 2-8 Znajdowanie liczb pierwszych w przedziale

### 3. Blokady czytelników i pisarzy

Blokady typu wzajemne wykluczanie są zbyt restrykcyjne co prowadzi do ograniczenia równoległości i spadku efektywności aplikacji.

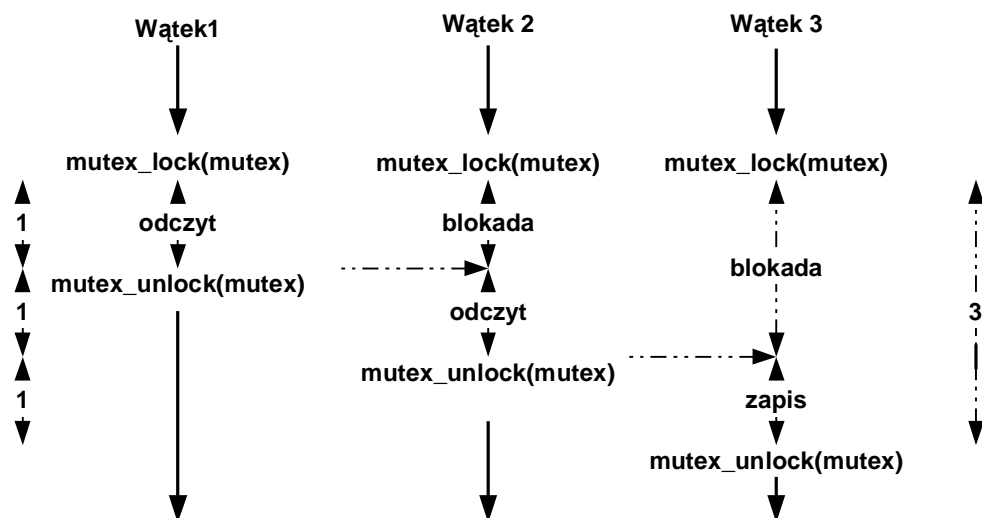
Można je osłabić wiedząc czy przeprowadzana jest operacja odczytu czy zapisu. Implementują to blokady czytelników i pisarzy (ang. *Readers writers locks*).

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.

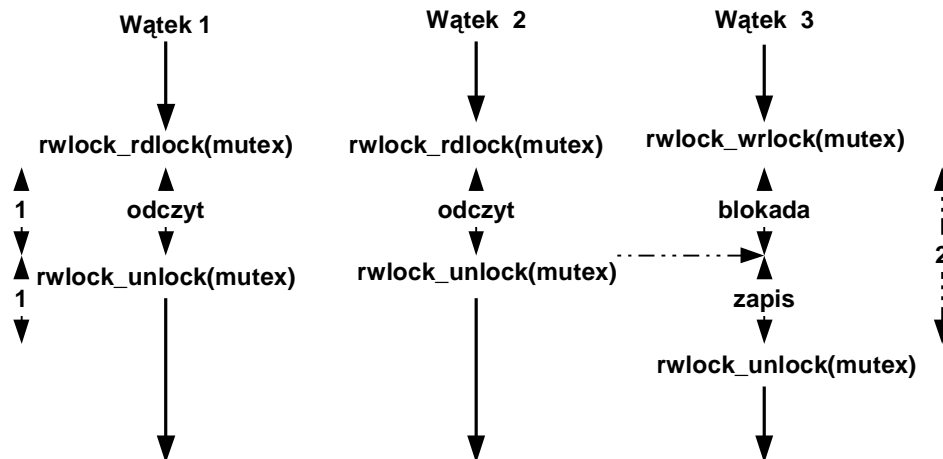
Stan blokady:

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



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





Rys. 3-2 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 mają być domyślne

### 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 zdejmuję 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.

### Nieblokujące zajęcie blokady do zapisu

```
int pthread_rwlock_trywrlock(pthread_rwlock_t  
*rwlock)
```

Gdy blokada jest wolna następuje jej zajęcie do zapisu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego 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.

### Nieblokujące zajęcie blokady do odczytu

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t  
*rwlock)
```

Gdy blokada jest wolna lub zajęta do odczytu następuje jej zajęcie do odczytu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego 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

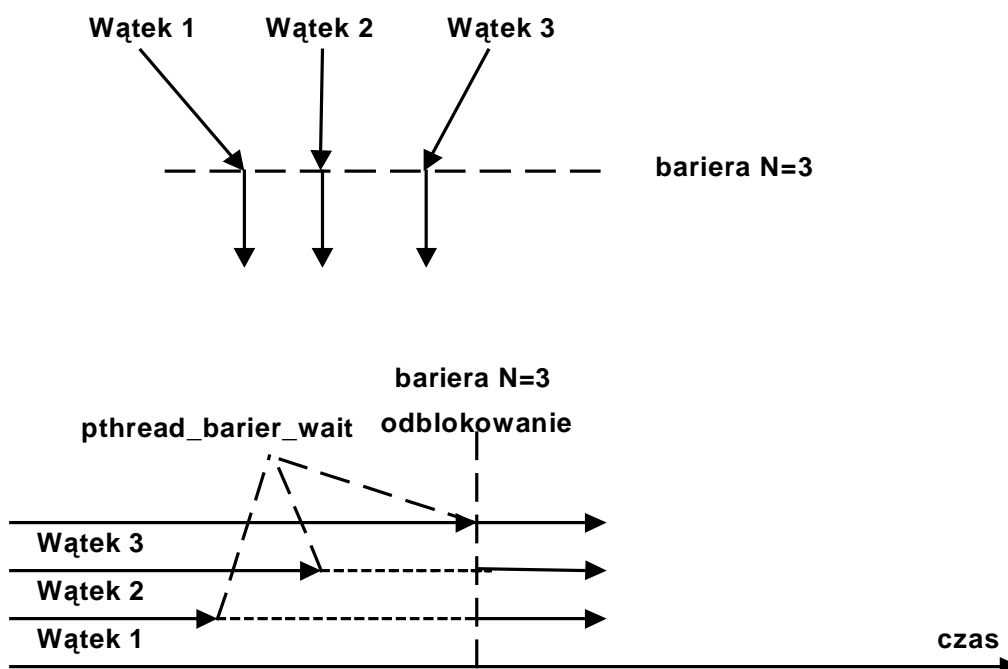
```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)
```

---

## Wątki

## 4. 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. 4-1 Działanie bariery

### Inicjacja bariery

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

**barrier**     Zadeklarowana zmienna typu `pthread_barrier_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\_barrier\_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\_barrier\_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 4-1 Przykład użycia bariery

## 5. 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.

## 6. Wątki w środowisku wieloprocessorowym

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.

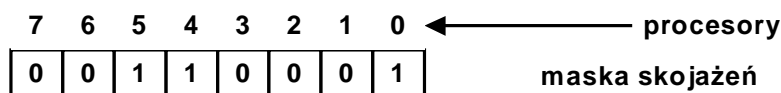
### Maska skojarzeń

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. 6-1 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:

<code>thread</code>	Identyfikator wątku
<code>cpusetsize</code>	Długość maski skojarzeń w bajtach
<code>cpuset</code>	Maska skojarzeń



Typ `cpu_set_t` – maska bitowa długości do 1024

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

Tabela 6-1 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)
```

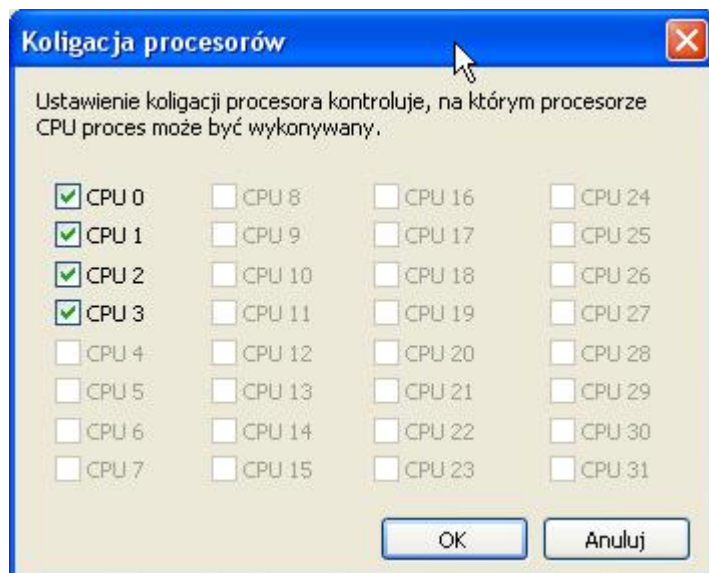
```
int pthread_attr_getaffinity_np(pthread_attr_t *attr,  
    size_t cpusetsize, cpu_set_t *cpuset)
```

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 6-1 Ustawianie i testowanie maski skojarzeń – dozwolone procesory 0 do 7



Rys. 6-2 Koligacja procesorów w Windows

## 7. Literatura:

- [1] Love Robert, Linux programowanie systemowe, Helion 2014
- [2] Manual systemu Linux, <http://www.kernel.org/doc/man-pages>