THREAD'S FUN, CZYLI SYNCHRONIZACJA mh TEX

W odniesieniu do wątków, podobnie jak i procesów można stosować celem synchronizacji semafory ***POSIX*** aczkolwiek jest to sposób co najmniej niewygodny.
Standard ***IEEE POSIX*** począwszy od **1003.1c** (1995), dla potrzeb wątków implementuje semafory ***MUTEX***.
Są one reprezentowane za pośrednictwem typu mutex_t (**bits/pthreadtypes.h** włączany do **pthreads.h**). Przed użyciem semafor mutex musi być – oczywiście – zainicjowany, a - kiedy już będzie niepotrzebny - z pamięci.

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_init( pthread_mutex_t* mutex,
    const pthread_mutexattr_t* attr );
int pthread_mutex_destroy( pthread_mutex_t* mutex );
pthread_mutex_t mutex -identyfikowator semafora
const pthread_mutexattr_t* attr -wartość inicjująca, zwykle NULL
pthread_mutex_t* mutex = PTHREAD_MUTEX_INITIALIZER;
a efekt będzie ten sam, ponieważ w
#({ { 0, 0, 0, 0, 0, 0, { 0, 0 } } })
```

RETURN

```
0 -jeżeli operację zakończona sukcesem
```

ERROR

```
kod błędu
```

MECHANIZMY SYNCHRONIZACJI MUTEX

MUTEX z definicji jest semaforem binarnym a więc dwustanowym, przeznaczonych od ochrony zasobów udostępnianych na wyłączność.
Chęć dostępu do zasobu powinna być poprzedzona wywołaniem **pthread_mutex_lock()** a jego zwolnienie **pthread_mutex_unlock()**, co stanowi analogię **P()** (**wait()**) i **V()** (**signal()**) Dijkstry.

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
pthread_mutex_t *mutex - identyfikator zainicjowanego uprzednio mutex'a
```

RETURN

```
0 -jeżeli wywołanie zakończone sukcesem
```

ERROR

```
kod błędu
```

Funkcja **pthread_mutex_trylock()** dokonuje dostępności zasobu bez wprowadzania blokady, aczkolwiek generuje błąd

```
EBUSY
```

jeżeli zasób współdzielony jest niedostępny.

MECHANIZMY SYNCHRONIZACJI MUTEX

Zacznijmy od prostego przykładu Daniela Robbinsa (nieznacznie zmodyfikowanego) ilustrującego sytuację wyścigu w dostępie do zasobu – tutaj – będzie to zmienna globalna, w przypadku dwóch wątków: główny w **main()** a potomny wykona **thread()**.

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

int N=0;                    //...zmienna na której będziemy działać
void *thread( void *arg )
{
    int n,i;
    for ( i=0;i<5;i++ )
    {
        n = N;              //...pobieramy wartość zmiennej globalnej
        n++;                //...inkrementacja kopii lokalnej
        printf("thread() [%lu]\n",(unsigned long)pthread_self());
        fflush( stdout );
        sleep( 1 );         //...czekamy 1 sekundę
        N = n;              //...przypisanie wartości zmiennej globalnej
    }
    pthread_exit( NULL );
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Ciało funkcji **main()** przedstawia się następująco.

```
int main( void )
{
    pthread_t tid;
    int i;
    //...tworzymy jeden wątek potomny (może trochę niefortunne określenie)
    if( pthread_create( &tid,NULL,thread,NULL ) )
    { perror( "...pthread_create()..." ); exit( 1 ); }
    for ( i=0;i<5;i++)
    {
        N--;           //...dekrementacja globalnej, w wątku głównym
        printf( "main()   [%lu]\n",(unsigned long)pthread_self() );
        fflush( stdout );
        sleep( 1 );    //...czekamy 1 sekundę
    }
    if( pthread_join( tid,NULL ) )
    { perror( "...pthread_join()..." ); exit( 2 ); }

    printf( "\nglobalnie N=%d, po wykonaniu 5+5 iteracji\n",N );
    return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Konieczna będzie konsolidacja z **libpthread.so**, czyli jeżeli zapiszemy pod pod nazwą **zero.c**

```
$ gcc -Wall zero.c -o zero -lpthread
```

Efekt wykonanie jest następujący

```
$ ./zero
main()   [47254501374720]
thread() [1082132800]
main()   [47254501374720]
thread() [1082132800]
main()   [47254501374720]
thread() [1082132800]
main()   [47254501374720]
thread() [1082132800]
main()   [47254501374720]
thread() [1082132800]
globalnie N=4, po wykonaniu 5+5 iteracji
```

czyli na pozór wszystko jest jak być powinno, oprócz wyniku końcowego:

jeżeli wartością początkową **N** było **0**,

wątek w **main()** pięciokrotnie wykonał dekrementację,

wątek w **thread()** pięciokrotnie wykonał inkrementację,

to wynik powinien być, ponownie

```
0
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Prześledźmy wykonanie wątków:

- proces jest uruchamiany N = 0
- zgodnie z komunikatem **`main()`** wykonuje **`N--`** (czyli jest **`-1`**) i zasypia na **`1`** sekundę;

następnie:

- sterowanie uzyskuje **`thread()`** pobiera pobiera **`N`** i przypisuje **`n`** (czyli jest**`-1`**), wykonuje **`n++`** (czyli w **`n`** jest **`0`**) i zasypia na **`1`** sekundę;
- w tym momencie budzi się **`main()`** i wykonuje wykonuje **`N--`** (czyli jest **`-2`**) i zasypia;
- budzie się **`thread()`** i wykonuje przypisanie **`N=n`**, więc podstawia **`0`** do **`N`**, i rozpoczynając kolejną iterację **`n`** będzie także **`0`**, więc po **`n++`** będzie miał w **`n`** wartość **`1`**, i zasypia;
- powraca **`mian()`** mając w zmiennej globalnej **`0`**, wykonuje **`N--`** (czyli jest **`-1`**) i zasypia;
- ale **`thread()`** odzyskuje sterowanie i podstawia pod **`N`** wartość **`1`**, następnie pobiera tę wartość i inkrementuje w zmiennej lokalnej **`n`** (uzyskuje w ten sposób **`2`**);

proces ten jest kontynuowany i w efekcie finalnym uzyskujemy

```
4
```

zamiast

```
0
```

bowiem konieczne byłoby tutaj zastosowania semafora zamykającego w niepodzielnym bloku – mimo i wbrew relacjom czasowym – działań realizowanych przez oba wątki.

MECHANIZMY SYNCHRONIZACJI MUTEX

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

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;  //...konieczna deklaracja
int N=0;

void *thread( void *arg )
{
    int n;
    int i;
    for ( i=0;i<5;i++ )
    {
        pthread_mutex_lock( &mutex );              //...zgłoszenie
        n = N; n++;
        printf( "thread() [%lu]\n",(unsigned long)pthread_self() );
        fflush( stdout );
        sleep( 1 ); N = n;
        pthread_mutex_unlock( &mutex );            //...i zwolnienie
    }
    pthread_exit( NULL );
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

```
int main( void )
{
    pthread_t tid;
    int i;
    if( pthread_create( &tid,NULL,thread,NULL ) )
    { perror( "...pthread_create()..." ); exit( 1 ); }
    for ( i=0; i<5; i++)
    {
        pthread_mutex_lock( &mutex );
        N--;
        printf( "main()   [%lu]\n",(unsigned long)pthread_self() );
        fflush( stdout );
        sleep( 1 );
        pthread_mutex_unlock( &mutex );
    }
    if( pthread_join( tid,NULL ) )
    { perror( "...pthread_join()..." ); exit( 2 ); }
    if( pthread_mutex_destroy( mutex) )
    { perror( "...pthread_mutex_destroy()..." ); exit( 3 ); }
    printf( "\nglobalnie N=%d, po wykonaniu 5+5 iteracji\n",N );
    return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

No a efekt wykonania

```
$ ./tzero
main()   [47257458977536]
main()   [47257458977536]
main()   [47257458977536]
main()   [47257458977536]
main()   [47257458977536]
thread() [1082132800]
thread() [1082132800]
thread() [1082132800]
thread() [1082132800]
thread() [1082132800]
```

globalnie N=0, po wykonaniu 5+5 iteracji

czyli taki jak należałoby oczekiwać.

Natomiast wniosek bardziej ogólny jest taki, że w przypadku wątków należy zwracać baczną uwagę na ewentualne skutki uboczne do zmiennych globalnych, które zawsze są "współdzielone" między wątkami.

MECHANIZMY SYNCHRONIZACJI MUTEX

Jak wiadomo suma kolejnych liczb naturalnych wyraża się

$$n=1,2,3,\ldots,n \quad \frac{n\cdot\left(n+1\right)}{2}$$

Przygotujemy program, który wykorzystując wątki oblicza sumę tego rodzaju ciągu.

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

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
int N=0; //...zmienna służąca przechowywaniu wartości sumy

void *thread( int id )              //...funkcja wątku
{
    pthread_mutex_lock( &mutex );  //...P() aby zapewnić
    {                              //   wątkowi wyłączność w momencie
        N += id;                   //   kiedy odwołuje się do
    }                              //   zmiennej globalnej
    pthread_mutex_unlock( &mutex );//...V() i zwalniamy
    pthread_exit( NULL );
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

```
int main( int argc,char** argv )
{
    pthread_t* tid;
    int        i,n;
    if( argc>1 )
    {
        sscanf( argv[1],"%d",&n );
        if( n<2 ){ printf( "...n<2, przyjęto n=2...\n" ); n = 2; }
        //...tworzymy tablicę w której będziemy przechowywać id wątków
        //   aby w przyszłości wykonać pthread_join()
        if( !(tid=(pthread_t*)calloc( (size_t)n,sizeof( pthread_t) ) ) )
        { perror( "...calloc()..." ); exit( 2 ); }

        for( i=0;i<n;i++ ) //...uruchamiamy wątki
        {
            if( pthread_create( (tid+i), NULL,(void*(*)(void*))thread, (void*)(i+1)) )
            {
                free( (void*)tid );
                perror( "...pthread_create()..." );
                exit( 2 );
            }
        }
```

MECHANIZMY SYNCHRONIZACJI MUTEX

W tym momencie pozostaje już tylko czekać aż wątki wykonają swoje zadania wykonując `pthread_join()`.

```
        for( i=0;i<n;i++ ) //...dołączamy w takim razie wątki
        {
            if( pthread_join( *(tid+i),NULL ) )
            {   //...tak na wszelki wypadek
                free( (void*)tid );
                perror( "...pthread_join()..." );
                exit( 2 );
            }
        }

        free( (void*)tid ); //...zwalniamy pamięć
        if( pthread_mutex_destroy( &mutex) )
        { perror( "...pthread_mutex_destroy()..." ); exit( 2 ); }
        printf( "|\n|%s%d%s%d\n|\n"," suma ciągu n=",n,
                " liczb naturalnych wynosi n*(n+1)/2=",N );
    }
    else //...gdyby wywołanie programu było niepoprawne
    { printf( "...wywołanie programu %s <ilość wątków>\n",argv[0] ); exit( 1 ); }

    return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Kompilacja

```
$ gcc -Wall sum.c -o sum -lpthread
```

i wykonanie

```
$ ./sum 100
|
| suma ciągu n=100 liczb naturalnych wynosi n*(n+1)/2=5050
|
```

istotnie

$$\frac{100\cdot\left(100+1\right)}{2}=\frac{100\cdot 101}{2}=\frac{10100}{2}=5050$$

Zwróćmy jeszcze uwagę na linę kodu

```
if( pthread_create( (tid+i), NULL,(void*(*)(void*))thread, (void*)(i+1)) )
```

konstrukcja

```
( void*( * )( void* ) )thread
```

służy konwersji typu – tym razem funkcji – która w wywołaniu `pthread_create()` powinno być

```
void* thread( void* arg)
```

a jest

```
void* thread( int arg )
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Bardzo często programy wykonują pewne działanie wobec ciągu identycznych elementów (zwykle zestawów danych). Niejako na bazie wcześniejszego, przyjrzyjmy się jak można zwiększyć efektywność przetwarzania wykorzystując w tym celu wątki.

```
//...zaczynamy od standardowych, w tym przypadku, deklaracji
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

//...inicjujemy domyślnie semafor binarny MUTEX
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

//...zmienna X będzie tablicą, której sumę elementów wyznaczymy
//   wartość sumy będzie pamiętana w zmiennej sum (wartość początkowa-oczywiście-0)
double  *X,sum=0.0;

//...zmienne globalne n i m posłużą zapamiętaniu rozmiaru tablicy i ilości wątków,
odpowiednio
unsigned long int n,m;
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Funkcja wątku może tutaj przedstawiać się następująco

```
void *thread( unsigned long int id )
{
    unsigned long int i;
    double x;

    for( i=id,x=0.0;i<n;i+=m )
    {
        x += *(X+i);
    }

    pthread_mutex_lock( &mutex );  //...powiedzmy że przypisanie wartości
    {                              //   zmiennej globalnej wykonamy
        sum += x;                  //   przy wyłącznym dostępie
    }
    pthread_mutex_unlock( &mutex );

    pthread_exit( NULL );
}
```

Zatem każdy wątek odwoływać się będzie tylko do części elementów, wyznaczając – tutaj sumę cząstkową.

MECHANIZMY SYNCHRONIZACJI MUTEX

W takim razie poszczególne wątki będą się odwoływać do

`0, 0+m, 0+m+m, 0+m+m+m,...`

`1, 1+m, 1+m+m, 1+m+m+m,...`

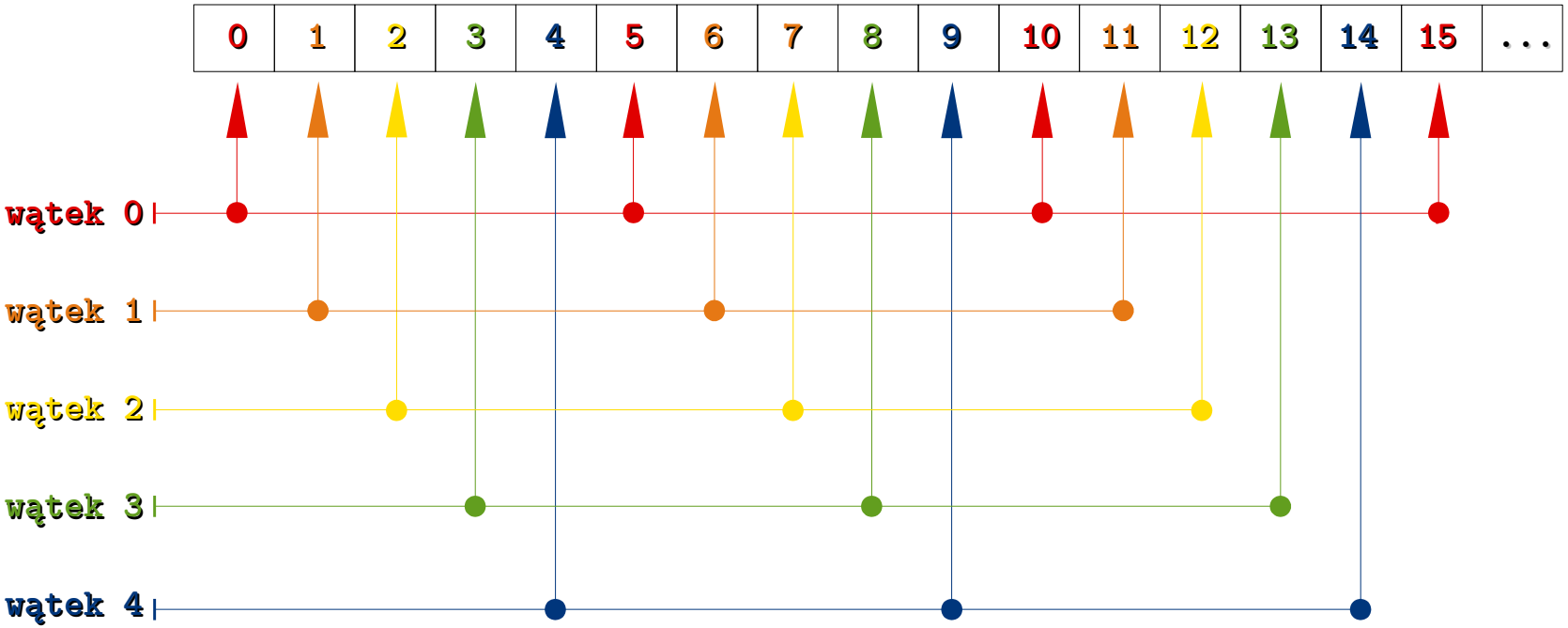
i tak dalej. Załóżmy, że mamy

`m`=5 wątków

odpowiednio

wątek 0 **wątek 1** **wątek 2** **wątek 3** **wątek 4**

wówczas odwołania do elementów tablicy mogą przebiegać jak na schemacie niżej.



MECHANIZMY SYNCHRONIZACJI MUTEX

W części głównej programu będziemy mieli

```
int main( int argc,char** argv )
{
    pthread_t* tid;
    unsigned long int i;

    if( argc>2 ) //...sprawdzamy czy wywołanie programu jest poprawne
    {
        /...pobieramy ilość wątków
        sscanf( argv[1],"%lu",&m );
        if( m<2  ){ printf( "...m<2, przyjęto m=2...\n" ); m=2; }
        //...alokacja tablicy w której pamiętać będziemy ID poszczególnych wątków
        tid = (pthread_t*)calloc( (size_t)m,sizeof( pthread_t) );
        //...pobieramy rozmiar tablicy X
        sscanf( argv[2],"%lu",&n );
        if( n<10 ){ printf( "...n<10,przyjęto n=1000...\n" ); n=1000; }
        //...alokacja tablicy i inicjowanie wartości
        if( !(X=(double*)calloc( (size_t)n,sizeof( double ) ) ) )
        {  perror( "...calloc()..." ); exit( 2 ); }
        for( i=0;i<n;i++ ){ *(X+i) = (double)(i+1); }
```

MECHANIZMY SYNCHRONIZACJI MUTEX

```
    for( i=0;i<m;i++ ) //...uruchamiamy zatem poszczególne wątki
    {
        if( pthread_create( (tid+i),NULL,( void*( * )( void* ) )thread,(void*)i) )
        {   free( (void*)tid ); free( (void*)X );
            perror( "...pthread_create()..." ); exit( 2 );
        }
    }
    for( i=0;i<m;i++ ) //...i dołączamy
    {
        if( pthread_join( *(tid+i),NULL ) )
        {   free( (void*)tid ); free( (void*)X );
            perror( "...pthread_join()..." ); exit( 2 );
        }
    }
    free( (void*)tid ); free( (void*)X );
    if( pthread_mutex_destroy( &mutex) )
    { perror( "...pthread_mutex_destroy()..." ); exit( 2 ); }
    printf( "|\n|%s%f\n|\n","suma elementów tablicy X wynosi ",sum );
    }
    else{ printf(   "...wywołanie programu %s
                    <ilość wątków> <rozmiar tablicy>\n",argv[0] ); exit( 1 ); }
    return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

W bardzo wielu przypadkach wykonanie wątku, czy kontynuacja wykonanie, musi być uzależniona od spełniania pewnego warunku. Oczywiście można by prowadzić ciągłe sprawdzanie instrukcją **if(...){...}** byłoby to jednak posunięcie dość nieefektywne. Znacznie lepiej w tym celu wykorzystać w tym celu zmienne warunkowe **pthread_cond_t**.

SYNOPSIS

```
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init( pthread_cond_t *cond, pthread_condattr_t *cond_attr );
int pthread_cond_destroy( pthread_cond_t *cond );
```

RETURN

0 -*jeżeli zakończono sukcesem*

ERROR

kod błędu

Zmienna warunkowa, która de facto jest unią

```
typedef union
{
    struct
    {
        int __lock; unsigned int __futex;
        __extension__ unsigned long long int __total_seq,__wakeup_seq,__woken_seq;
        void *__mutex; unsigned int __nwaiters,__broadcast_seq;
    } __data;
    char __size[__SIZEOF_PTHREAD_COND_T]; __extension__ long long int __align;
} pthread_cond_t;
```

MECHANIZMY SYNCHRONIZACJI MUTEX

W sposobie inicjowania i usuwania zmiennej warunkowej jest nieprzypadkowa analogia do zmiennych ***MUTEX***.

Po zainicjowaniu danej zmiennej warunkowej, wywołując **`pthread_cond_wait()`** wątek przechodzi w stan oczekiwania aż pojawi się sygnał o spełnieniu warunku wygenerowany przez **`pthread_cond_signal()`** albo **`int pthread_cond_broadcast()`** , przy czym ta druga funkcja zwalnia wszystkie wątki oczekujące w kolejce związanej z danym warunkiem. Wygenerowanie sygnału w przypadku kiedy nie ma wątków oczekujących nie powoduje żadnego skutku.

SYNOPSIS

```
#include <pthread.h>
int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex );
int pthread_cond_signal( pthread_cond_t *cond );
int pthread_cond_broadcast( pthread_cond_t *cond );
```

RETURN

```
0 -jeżeli zakończono sukcesem
```

ERROR

```
kod błędu
```

Zauważmy że – co logiczne – funkcja **`int pthread_cond_wait()`** wymaga zdefiniowanego semafora binarnego ***MUTEX***.

MECHANIZMY SYNCHRONIZACJI MUTEX

Najprostszy przykład (schemat) użycia zmiennej warunkowej.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int go = 0;             /...zmienna sterująca wykonaniem (określa warunek)
pthread_cond_t  cond    = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex   = PTHREAD_MUTEX_INITIALIZER;

int main( int argc, char *argv[] )
{
   pthread_t cid,pid;

   if( pthread_create( &cid,NULL,consumer,NULL ) )
   { perror( "...pthread_create()..." ); exit( 1 ); }
   if( pthread_create( &pid,NULL,producer,NULL ) )
   { perror( "...pthread_create()..." ); exit( 1 ); }
   if( pthread_join( cid,NULL ) )
   { perror( "...pthread_join()..." ); exit( 1 ); }
   if( pthread_join( pid,NULL ) )
   { perror( "...pthread_join()..." ); exit( 1 ); }
   return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

```
void* producer( void* arg )
{
   printf ("[%lu] producent: ???\n",(unsigned long)pthread_self() );
   sleep (1);

   pthread_mutex_lock( &mutex );
   go = 1;
   pthread_cond_signal( &cond );
   pthread_mutex_unlock( &mutex );

   pthread_exit( NULL );
}

void* consumer( void* arg )
{
   printf ("[%lu] konsument: czeka...\n",(unsigned long)pthread_self() );

   pthread_mutex_lock (&mutex);
   while( !go ){ pthread_cond_wait( &cond,&mutex ); }
   pthread_mutex_unlock (&mutex);
   printf ("[%lu] konsument: kontynuacja\n",(unsigned long)pthread_self() );

   pthread_exit( NULL );
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Rozpatrzmy przykład kiedy mamy dwa wątki dla pierwszy **worker()** wykonuje określone działania na ustalonej zmiennej, zaś **watch()** czeka aż zmienna to osiągnie odpowiednią wartość.

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

pthread_mutex_t   mutex;
pthread_cond_t    cond;
unsigned long int n=0;          //...monitorowana zmienna
const unsigned long int no=10; //...pożądana wartość

int main( int argc,char** argv )
{
    pthread_t tid[2];

    pthread_mutex_init( &mutex,NULL ); pthread_cond_init ( &cond,NULL  );
    pthread_create((tid+0),NULL,worker,NULL); pthread_create((tid+1),NULL,watch,NULL);
    pthread_join( *(tid+1),NULL ); pthread_join( *(tid+0),NULL );
    pthread_mutex_destroy( &mutex ); pthread_cond_destroy(  &cond  );

    printf( "[%lu] main(): stop\n",(unsigned long)pthread_self() );

   return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Funkcja dla czekającego i roboczego przedstawia się następująco:

```
void* watch( void *arg )
{
    printf( "[%lu] watch(): start\n",(unsigned long)pthread_self() );
    pthread_mutex_lock( &mutex );
    while( n<no ){ pthread_cond_wait( &cond,&mutex ); }
    printf( "[%lu] watch(): V() %lu\n",(unsigned long)pthread_self(),n );
    pthread_mutex_unlock( &mutex );
    printf( "[%lu] watch(): stop\n",(unsigned long)pthread_self() );
    pthread_exit( NULL );
}

void* worker( void *arg )
{
    for( n=0;n<25;n++ )
    {
        //pthread_mutex_lock(&mutex);
        printf( "[%lu] worker(): %lu\n",(unsigned long)pthread_self(),n);
        if( !(n-no) ){ pthread_cond_broadcast( &cond ); break; }
        //...równoważnie if( !(n-no) ){ pthread_cond_signal( &cond ); }
        //pthread_mutex_unlock( &mutex );
    }
    pthread_exit( NULL );
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Efekt wykonania jest następujący

```
$ ./cond
[1082132800] worker(): 0
[1082132800] worker(): 1
[1082132800] worker(): 2
[1082132800] worker(): 3
[1082132800] worker(): 4
[1082132800] worker(): 5
[1082132800] worker(): 6
[1082132800] worker(): 7
[1082132800] worker(): 8
[1082132800] worker(): 9
[1082132800] worker(): 10
[1090525504] watch(): start
[1090525504] watch(): V() 10
[1090525504] watch(): stop
[47505656392448] main(): stop
```

MECHANIZMY SYNCHRONIZACJI MUTEX

W niektórych przypadkach zachodzić może potrzeba synchronizacji wszystkich wątków procesu z pewnym punktem wykonania kodu. Służą temu bariery ***POSIX***. Barierę wprowadzają i likwidują wywołania dwóch funkcji **pthread_barrier_init()** oraz **pthread_barrier_destroy()**.

SYNOPSIS

```
#include <pthread.h>
int pthread_barrier_init( pthread_barrier_t* barrier,
    const pthread_barrierattr_t* attr, unsigned count );
int pthread_barrier_destroy( pthread_barrier_t *barrier );
pthread_barrier_t* barrier -zmienna identyfikująca barierę
pthread_barrierattr_t* attr -atrybuty bariery (domyślnie NULL)
unsigned count -ilość wątków która musi się odwołać do bariery
    aby nastąpiło zwolnienie
int pthread_barrier_wait( pthread_barrier_t *barrier );
```

RETURN

```
0 -jeżeli sukces albo kod błędu
```

Ustawiając barierę **barrier** określamy licznik **count**. W trakcie wykonania wątków bariera będzie obowiązywać tak długo – tak długo wątki nie zostaną zwolnione – aż ilość wywołań **pthread_barrier_wait()** odwołań do danej bariery osiągnie **count**.
Bariera nie wymaga dodatkowych form blokad, jak semafory **MUTEX**.

MECHANIZMY SYNCHRONIZACJI MUTEX

Teraz przykład prezentujący sposób wprowadzenia i użycia bariery synchronizacyjnej.

```
//...zaczynamy od standardowego zestawu plików włączanych
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>                     //...tylko aby wywołać sleep()
#include <pthread.h>

pthread_barrier_t barrier;              //...zmienna identyfikująca barierę
unsigned count;                         //...licznik odwołań do bariery

void *thread( void *arg )               //...funkcja wątku, bardzo prosta
{                                       //   tylko komunikat diagnostyczny i ...
   printf( "[%lu] thread wait signal...!\n",(unsigned long)pthread_self() );
   pthread_barrier_wait( &barrier );    //...wait() względem bariery
   pthread_exit( NULL );
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

```
int main( int argc,char** argv )
{
   pthread_t* tid;
   int        i,n;

   if( argc>1 )
   {
      sscanf( argv[1],"%d",&n );
      if( n<2 ){ printf( "...n<2, przyjęto n=2...\n" ); n = 2; }
      count = n+1; //...licznik bariery ustawiamy na n wątków + 1 !
      if( !(tid=(pthread_t*)calloc( (size_t)n,sizeof( pthread_t) ) ) )
      { perror( "...calloc()..." ); exit( 2 ); }
      pthread_barrier_init( &barrier,NULL,count );

      for( i=0;i<n;i++ )
      {
         if( pthread_create( (tid+i), NULL,thread,NULL ) )
         {
            free( (void*)tid );
            perror( "...pthread_create()..." );
            exit( 2 );
         }
      }
```

MECHANIZMY SYNCHRONIZACJI MUTEX

```
        //... no i w tym miejscu mamy wątek (n+1) !
        printf( "[%lu] main()...?\n",(unsigned long)pthread_self() );
        sleep( 1 );
        pthread_barrier_wait( &barrier ); //...teraz dopiero będzie zwolniona
        printf( "[%lu] main()...done\n",(unsigned long)pthread_self() );

        for( i=0;i<n;i++ ) //...łączymy wątki
        {
            if( pthread_join( *(tid+i),NULL ) )
            {free( (void*)tid );perror( "...pthread_join()..." );exit( 2 );}
        }

        //...i końcowe porządki
        free( (void*)tid );
        if( pthread_barrier_destroy( &barrier ) )
        { perror( "...pthread_barrier_destroy()..." ); exit( 2 ); }

    }
    else{ printf("...wywołanie programu %s <ilość wątków>\n",argv[0] );exit( 1ˆ ); }

    return 0;
}
```

MECHANIZMY SYNCHRONIZACJI MUTEX

Przykładowe wykonanie dla 10 wątków

```
$ ./barrier 10
[1082132800] thread wait signal...!
[1090525504] thread wait signal...!
[1098918208] thread wait signal...!
[47732824382208] main()...?
[1107310912] thread wait signal...!
[1115703616] thread wait signal...!
[1124096320] thread wait signal...!
[1132489024] thread wait signal...!
[1140881728] thread wait signal...!
[1149274432] thread wait signal...!
[1157667136] thread wait signal...!
[47732824382208] main()...done
```

Zwróćmy uwagę, że wprowadzenie bariery nie wymusza żadnych relacji czasowych wykonania – wątki będą się wykonywać ale żaden z nich nie przekroczy wywołania

```
pthread_barrier_wait( &barrier );
```

dopóki licznik `count` nie osiągnie wartości zadanej w trakcie inicjowania bariery, czyli w

```
pthread_barrier_init( &barrier,NULL,count );
```