Uniwersytet Bielsko-Bialski

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 9

mutex

GRUPA: 2B / SEMESTR: 5 / ROK: 3

Igor Gawłowicz / 59096

Cel ćwiczenia

Celem ćwiczenia jest eksploracja i zrozumienie mechanizmów synchronizacji wątków w systemie operacyjnym, a także zastosowanie semaforów, zmiennych warunkowych, i barier w celu zarządzania dostępem do współdzielonych zasobów między wątkami. Ćwiczenie polega na implementacji kodu w języku C, który wykorzystuje różne techniki synchronizacji wątków takie jak mutexy, zmienne warunkowe i bariery, aby kontrolować dostęp do wspólnych zasobów.

Przebieg ćwiczenia

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.

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.

Funkcja pthread_mutex_trylock() dokonuje dostępności zasobu bez wprowadzania blokady, aczkolwiek generuje błąd **EBUSY** jeżeli zasób współdzielony jest niedostępny.

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 );
}

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;
}
```

Co ciekawe pomimo tego że zarówno wątek jak i proces, wykonują tą samą lecz przeciwną operację wynik nie wychodzi równy 0 tylko jak poniżej

```
$ ./zero
main() [140390545930048]
thread() [140390545925888]
main() [140390545930048]
thread() [140390545925888]
main() [140390545930048]
thread() [140390545925888]
main() [140390545930048]
thread() [140390545925888]
thread() [140390545925888]
main() [140390545930048]
```

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.

```
#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 );
}

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;
}
```

Tym razem otrzymamy już oczekiwany wynik

```
$ ./tzero
main() [133727424735040]
main() [133727424735040]
thread() [133727424730880]
thread() [133727424730880]
thread() [133727424730880]
thread() [133727424730880]
thread() [133727424730880]
main() [133727424735040]
main() [133727424735040]
main() [133727424735040]
```

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.

Jak wiadomo suma kolejnych liczb naturalnych wyraża się

n=1,2,3,....,n

n*(n+1)/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 );
}

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 );
            }
        }
        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;
}
```

Po uruchomieniu programu otrzymamy następujący wynik

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

ardzo 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;

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 );
}

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);
        }
        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;
}
```

Taki program po uruchomieniu zwraca

```
$ ./test 5 5000
|
|suma elementów tablicy X wynosi 12502500.000000
|
```

Możemy zrobić porównanie dwóch poprzednich programów pod względem czasu, gdzie dla pierwszej wersji podamy n=5000, a dla drugiej ilość wątków = 5 i n = 5000

```
zciwolvo@cloudshell:~ (glassy-courage-399021)$ time ./sum 5000
|
| suma ciągu n=5000 liczb naturalnych wynosi n*(n+1)/2=12502500
|

real    0m0.284s
user    0m0.057s
sys     0m0.275s
zciwolvo@cloudshell:~ (glassy-courage-399021)$ time ./test 5 5000
|
|suma elementów tablicy X wynosi 12502500.000000
|

real    0m0.003s
user    0m0.001s
sys     0m0.002s
```

Możemy zauważyć że przy użciu 5 wątków czas pracy jest znacznie krótszy, a wynik dokładnie taki sam.

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.

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.

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;

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 );
}

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;
}
```

Powyższy program da nam taki wynik:

```
$ ./cond1
[136582164489984] konsument: czeka...
[136582156097280] producent: ???
[136582164489984] konsument: kontynuacja
```

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

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 );
}

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;
}
```

Otrzymamy następujący output

```
$ ./cond2
[138991326258944] worker(): 0
[138991326258944] worker(): 1
[138991326258944] worker(): 2
[138991326258944] worker(): 3
[138991326258944] worker(): 4
[138991326258944] worker(): 5
[138991326258944] worker(): 6
[138991326258944] worker(): 7
[138991326258944] worker(): 8
[138991326258944] worker(): 9
[138991326258944] worker(): 10
[138991317866240] watch(): start
[138991317866240] watch(): V() 10
[138991317866240] watch(): stop
[138991326263104] main(): stop
```

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().

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.

```c
//...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 );
}

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 );
            }
        }
        //... 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( 1 );
    }
    return 0;
}
```

```
$ ./barrier 10
[136660141311744] thread wait signal...!
[136660116133632] thread wait signal...!
[136660107740928] thread wait signal...!
[136660099348224] thread wait signal...!
[136660020885248] thread wait signal...!
[136660012492544] thread wait signal...!
[136660141315904] main()...?
[136660124526336] thread wait signal...!
[136660132919040] thread wait signal...!
[136660004099840] thread wait signal...!
[136659981915904] thread wait signal...!
[136660141315904] 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);

Wnioski

1. **Semafor MUTEX**:
    - Mutexy są używane do zapewnienia wyłącznego dostępu do współdzielonych zasobów w obrębie wątków.
    - Funkcje `pthread_mutex_lock()` i `pthread_mutex_unlock()` pozwalają na zablokowanie i odblokowanie mutexu, zapewniając, że tylko jeden wątek może jednocześnie korzystać z chronionego zasobu.

- Wykorzystanie mutexów eliminuje błędy wynikające z jednoczesnego dostępu wielu wątków do wspólnych zasobów, co prowadzi do konsystentnego stanu danych.

2. **Zmienne warunkowe**:

   - Zmienne warunkowe (`pthread_cond_t`) są przydatne, gdy wątek musi czekać na wystąpienie określonego warunku przed kontynuacją.
   - Funkcje `pthread_cond_wait()` i `pthread_cond_signal()` pozwalają wątkom na oczekiwanie na wystąpienie określonego warunku lub na sygnalizowanie innym wątkom, że warunek został spełniony.
   - Użycie zmiennych warunkowych eliminuje potrzebę ciągłego sprawdzania warunków w pętli, co prowadzi do bardziej wydajnego i przewidywalnego działania programu.

3. **Bariery**:

   - Bariery (`pthread_barrier_t`) są wykorzystywane do synchronizacji grupy wątków, które muszą poczekać, aż wszystkie osiągną określony punkt w kodzie przed kontynuacją.
   - Funkcje `pthread_barrier_init()` i `pthread_barrier_wait()` pozwalają na inicjalizację bariery i oczekiwanie na osiągnięcie przez wszystkie wątki punktu synchronizacji.
   - Użycie barier umożliwia kontrolowanie przebiegu działania wielu wątków, czekając aż wszystkie osiągną wspólny punkt synchronizacji.

4. **Zarządzanie zasobami w wielowątkowym środowisku**:

   - Korzystanie z mechanizmów synchronizacji takich jak mutexy, zmienne warunkowe i bariery pozwala na kontrolę dostępu do wspólnych zasobów.
   - Poprawne zarządzanie synchronizacją wątków jest kluczowe dla zapewnienia spójności danych i uniknięcia błędów wynikających z jednoczesnego dostępu wielu wątków do tych samych zasobów.

5. **Optymalizacja wydajności poprzez wielowątkowość**:

   - Wprowadzenie wielowątkowości może znacząco poprawić wydajność programu, zwłaszcza gdy wiele zadań może być wykonywanych równolegle.
   - Optymalne wykorzystanie wątków, poprzez efektywne zarządzanie synchronizacją, może przyspieszyć przetwarzanie i zwiększyć efektywność systemu.