Uniwersytet Bielsko-Bialski

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 10

Wybrane problemy

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

Igor Gawłowicz / 59096

Cel ćwiczenia

- Symulacja procesu produkującego i konsumującego dane.
- Zastosowanie semaforów do synchronizacji dostępu do współdzielonych zasobów (bufora).
- Symulacja problemu dostępu czytelników i pisarzy do współdzielonych zasobów.
- Zastosowanie semaforów do kontrolowania dostępu.
- Symulacja problemu znanego jako "problem ucztujących filozofów".
- Zastosowanie semaforów do uniknięcia zakleszczeń i równomiernego dostępu do zasobów.

Przebieg ćwiczenia

Piewrszy fragment kodu przedstawia nam przykład problemu typu producent/consumer, polegający na producencie, który tworzy jakieś wartości i konsumencie który te wartości wykorzystuje.

W najprostszym wariancie rozwiązanie zadania można byłoby sobie wyobrazić w następujący sposób:

1. program pobierze z linii komend argument określający ilość liczb pierwszych jaka będzie wygenerowana
2. producent i konsument stanowić będą dwa wątki o funkcjach producer() i consumer(), odpowiednio
3. wymiana danych między producentem a konsumentem zachodzić będzie przez bufor globalny buffer (tutaj jedno-elementowy);
4. z pewnością dostęp do bufora powinien być operacją wyłączną, w takim razie wprowadzimy smafor binarny mutex.

Jest prostym przykładem programu wielowątkowego, który symuluje działanie producenta i konsumenta. Wykorzystuje wątki do wykonywania równoległych zadań. Producent oblicza wartości i umieszcza je w globalnym buforze, podczas gdy konsument pobiera te wartości z bufora i wyświetla je. Semafor mutex jest wykorzystywany do synchronizacji dostępu do bufora między wątkami, zapewniając bezpieczeństwo dostępu do danych współdzielonych między wątkami. Jednak kod ten może wymagać poprawek w celu zapewnienia pełnej poprawności i bezpieczeństwa w środowisku wielowątkowym.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define ENDLESS 1 //...dla sterowania pętli producenta
//...semafor mutex celem zapewnienia dostępności wyłącznej, inicjowany domyślnie
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
unsigned long int buffer; //...bufor globalny wymiany danych
int n,no; //...zmienne sterujące wykonaniem konsumenta

unsigned long int produce( void )
{
    /* Formuła Jarosława Wróblewskiego i Jean-Charles'a Meyrignac'a (2006) */
    return (42*n*n*n + 270*n*n - 26436*n + 250703);
}

void append( unsigned long int v )
{
    buffer = v;
```

```
    return;
}

unsigned long int take( void )
{
    return buffer;
}

void* producer( void *arg )
{
    unsigned long int v;
    //...komunikat diagnostyczny
    printf( "[%lu] producer, start\n",(unsigned long)pthread_self() );
    //...w pętli bez końca, bo łączeniu podlegają konsumenci
    while( ENDLESS )
    {
        pthread_mutex_lock( &mutex );
        v = produce(); //...to wywołanie jest bezpieczne ale może być
problematyczne
        append( v ); //...i dodajemy do bufora (tu: wstawiamy właściwie)
        pthread_mutex_unlock( &mutex );
    }
    //...de facto ten fragment nie będzie nigdy wykonany
    printf( "[%lu] producer, stop\n",(unsigned long)pthread_self() );
    pthread_exit(NULL);
}

void *consumer( void* arg )
{
    unsigned long int v; //...typ musi odpowiadać temu co jest pobierane
    //...komunikat diagnostyczny
    printf( "[%lu] consumer, start\n",(unsigned long)pthread_self() );
    //...i zaczynamy
    for( n=0;n<no;n++ )
    {
        pthread_mutex_lock( &mutex );
        v = take();
        pthread_mutex_unlock( &mutex );
        printf("%d -> %lu\n",n,v );
    }
    printf( "[%lu] consumer, stop\n",(unsigned long)pthread_self() );
    pthread_exit(NULL);
}

int main( int argc,char** argv )
{
    pthread_t _pid; //...ID wątki producenta
    pthread_t _cid; //...ID wątki konsumenta
    void *producer( void* ); //...ewentualnie deklaracje nagłówkowe
    void *consumer( void* );
    if( argc>1 )
    {
        sscanf( argv[1],"%d",&no ); if( no>40 )
        {
```

```
            no=40;
        };
        pthread_create( &_cid,NULL,consumer,NULL );
        pthread_create( &_pid,NULL,producer,NULL );
        pthread_join( _cid,NULL ); //...łączone tylko wątki konsumenta (-ów)
    }
    else
    {
        printf( "...%s no= ???\n",argv[0] );
    }
    return 0;
}
```

Po kompilacji i uruchomieniu programu, możemy zaobserwować że wyniki nie zgadzają nam się z oczekiwaniami

```
./pc-1 10
[138865224574720] consumer, start
0 -> 0
1 -> 0
2 -> 0
3 -> 0
4 -> 0
5 -> 0
6 -> 0
7 -> 0
8 -> 0
9 -> 0
[138865224574720] consumer, stop
[138865216182016] producer, start
```

Problem polega na koordynacji między producentem a konsumentem, aby go rozwiązac możemy zastosować dodatkowy semafor.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define ENDLESS 1
unsigned long int buffer;
int n,no;
//...dwa semafory, odpowiednio dla:
pthread_mutex_t delay = PTHREAD_MUTEX_INITIALIZER; //...konsumenta
pthread_mutex_t exclude = PTHREAD_MUTEX_INITIALIZER; //...producenta

void append( unsigned long int v )
{
    buffer = v;
    return;
}
```

```
unsigned long int take( void )
{
    return buffer;
}

unsigned long int produce( void )
{
    return (42*n*n*n + 270*n*n - 26436*n + 250703);
}

void* producer( void *arg )
{
    unsigned long int v;
    printf( "[%lu] producer, start\n",(unsigned long)pthread_self() );
    while( ENDLESS )
    {
        pthread_mutex_lock(&exclude);//...żądanie wyłączności dostępu do bufora
        v = produce();
        append( v );
        pthread_mutex_unlock( &delay );//...ewentualne zwolnienie klienta
    }
    printf( "[%lu] producer, stop\n",(unsigned long)pthread_self() );
    pthread_exit(NULL);
}

void *consumer( void* arg )
{
    unsigned long int v;
    printf( "[%lu] consumer, start\n",(unsigned long)pthread_self() );
    for( n=0;n<no;n++ )
    {
        //...pytanie o dostępność bufora
        pthread_mutex_lock( &delay );
        //...jeżeli tak, to pobieramy daną
        v = take();
        //...ten moment możemy wykorzystać dla ewent. zwolnienia producenta
        pthread_mutex_unlock( &exclude);
        printf("%d -> %lu\n",n,v );
    }
    printf( "[%lu] consumer, stop\n",(unsigned long)pthread_self() );
    pthread_exit(NULL);
}

int main( int argc,char** argv )
{
    pthread_t _pid,_cid;
    if( argc>1 )
    {
        sscanf( argv[1],"%d",&no );
        if( no>40 )
        {
            no=40;
        };
```

```
        pthread_mutex_lock( &delay );
        pthread_create( &_cid,NULL,consumer,NULL );
        pthread_create( &_pid,NULL,producer,NULL );
        pthread_join( _cid,NULL );
    }
    else
    {
        printf( "...%s no= ???\n",argv[0] );
    }
    return 0;
}
```

Tym razem możemy zaobserwować że wyniki wydają się bardziej rzetelne

```
$ ./pc-3 10
[133131453089536] consumer, start
[133131444696832] producer, start
0 -> 250703
1 -> 224579
2 -> 224579
3 -> 174959
4 -> 151967
5 -> 130523
6 -> 110879
7 -> 93287
8 -> 77999
9 -> 65267
[133131453089536] consumer, stop
```

Zajmijmy się teraz kolejnym klasycznym problemem współbieżności readers/writers.

Załóżmy na wstępie:

1. będziemy mieli R czytelników i W pisarzy a każdy z nich dokona N cykli zapisu, po czym program zakończy działanie;
2. jest rzeczą naturalną, że wykorzystamy tu dwa semafory (w zupełności wystarczającymi będą binarne mutex);
3. sformułowanie rozwiązania może być bardzo różne, zależnie od zamierzonego efektu – tutaj przyjmiemy preferencję dla czytelników.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define R 4 //...ilość czytelników
#define W 2 //...ilość pisarzy
#define N 5 //...ilość wejść pisarza do czytelni
#define ENDLESS 1
pthread_mutex_t wmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rmutex = PTHREAD_MUTEX_INITIALIZER;
```

```c
int counter; //...zmienna monitorująca ilość czytelników w czytelni
void* writer( void* arg )
{
    int n; //...zmienna posłuży do zliczania ilości cykli pisarzy
    for( n=0;n<N;n++)
    {
        //...zgłoszenie żądania dostępu na wyłączność czytelni
        pthread_mutex_lock( &wmutex );
        printf( "[%lu] pisarz, start\n",(unsigned long)pthread_self() );
        sleep(1); //...wykonanie operacji przez pisarza (tu: pusta)
        printf( "[%lu] pisarz, stop\n",(unsigned long)pthread_self() );
        //...zwolnienie czytelni (semafora)
        pthread_mutex_unlock( &wmutex );
        sleep( n%2 ); //...aby nieco utrudnić koordynację
    }
    pthread_exit( NULL );
}

void* reader( void* arg )
{
    int n=0; //...wyłącznie po to aby utrudnić (sprawdzić) koordynację
    while( ENDLESS ) //...bez końca bo monitorujemy pisarzy
    {
        pthread_mutex_lock( &rmutex );
        counter++; //...wchodzi czytelnik
        //...ponieważ pisarz musi mieć czytelnię na wyłączność, stąd....
        if( counter==1 )
        {
            pthread_mutex_lock( &wmutex );
        }
        pthread_mutex_unlock( &rmutex );
        printf( "[%lu] czytelnik, start\n",(unsigned long)pthread_self() );
        sleep(1); //...odczyt, tutaj jako operacja pusta
        printf( "[%lu] czytelnik, stop\n",(unsigned long)pthread_self() );
        pthread_mutex_lock( &rmutex );
        counter--; //...czytelnik zwalnia czytelnię
        //...jeżeli czytelnia pusta, to możemy wpuścić (zwolnić) pisarza
        if( !counter )
        {
            pthread_mutex_unlock( &wmutex );
        }
        pthread_mutex_unlock( &rmutex );
        sleep( n%3 ); n++; //...żeby trochę utrudnić
    }
    pthread_exit(NULL);
}

int main( void )
{
    pthread_t rid[R];
    pthread_t wid[W];
    int t;
    counter = 0; //...zero czytelników w czytelni
    //...uaktywniamy wątki czytelników
```

```
    for( t=0;t<R;t++ )
    {
        pthread_create( (rid+t),NULL,reader,NULL );
    }
    //...uaktywniamy wątki pisarzy
    for( t=0;t<W;t++ )
    {
        pthread_create( (wid+t),NULL,writer,NULL );
    }
    //...łączymy pisarzy, ponieważ akurat ich działania monitorujemy
    for( t=0;t<W;t++ )
    {
         pthread_join( *(wid+t),NULL );
    }
    return 0;
}
```

Program po kompilacji i uruchomienia zwróci nam następujący wynik

```
$ ./rw
[134876250441472] czytelnik, start
[134876133062400] czytelnik, start
[134876258834176] czytelnik, start
[134876242048768] czytelnik, start
[134876250441472] czytelnik, stop
[134876133062400] czytelnik, stop
[134876250441472] czytelnik, start
[134876133062400] czytelnik, start
[134876242048768] czytelnik, stop
[134876258834176] czytelnik, stop
[134876258834176] czytelnik, start
[134876242048768] czytelnik, start
[134876250441472] czytelnik, stop
[134876258834176] czytelnik, stop
[134876133062400] czytelnik, stop
[134876242048768] czytelnik, stop
[134876233656064] pisarz, start
[134876233656064] pisarz, stop
[134876225263360] pisarz, start
[134876225263360] pisarz, stop
[134876258834176] czytelnik, start
```

Problem ten polega na tym, że pewna liczba czytelników i pisarzy dzieli wspólny zasób, w tym przypadku fikcyjną "czytelnię". Pisarze mogą zapisywać w czytelni, podczas gdy czytelnicy mogą tylko czytać, ale nie mogą zapisywać.

Opis logiki działania programu:

Program uruchamia R wątków reprezentujących czytelników i W wątków reprezentujących pisarzy. Każdy z pisarzy próbuje uzyskać dostęp do czytelni N razy, wyświetlając komunikat "pisarz, start" i "pisarz, stop", a

następnie przerywa na chwilę, aby utrudnić koordynację. Czytelnicy również próbują uzyskać dostęp do czytelni w pętli nieskończonej, wyświetlając komunikaty "czytelnik, start" i "czytelnik, stop" oraz wykonywując operację odczytu przez określony czas. Następnie zwalniają dostęp do czytelni.

Wyniki wydaje się zgodny z przewidywanym i możemy na podstawie niego zaobserwować to jak program radzi sobie z alokacją zasobów i dostępów do danych.

Kolejny przykład dotyczyć będzie problemu dining philosophers

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define N 5 //...pięciu filozofów
#define CYCLE 1 //...każdy powinien uzyskać CYCLE-krotnie pałeczki
pthread_mutex_t sticks[N]; //...semafor monitorujący dostęp do pałeczek

void* diner( void* arg )
{
    int v;
    int eating = 0;
    v = *((int*)arg) +1;
    while( eating<CYCLE )
    {
        printf( "[%d]...hmm...co ja to miałem\n",v );
        sleep( v/2);
        printf( "[%d]...głodny\n", v);
        pthread_mutex_lock( (sticks+v) );
        pthread_mutex_lock( (sticks + (v+1)%N ) );
        printf( "[%d]...obiad...start\n",v );
        eating++;
        sleep(1);
        printf( "[%d]...obiad...stop\n",v );
        pthread_mutex_unlock( (sticks+v) );
        pthread_mutex_unlock( (sticks + (v+1)%N ) );
    }
    pthread_exit(NULL);
}

int main( void )
{
    pthread_t tid[N];
    int n[N]; //...tylko w celu diagnostycznym (aby określić numer kolejny)
    int i;
    //...inicjujemy semafor monitorujący dostępność pałeczek
    for (i=0;i<N;i++)
    {
        pthread_mutex_init( (sticks+i), NULL );
    }
    for (i=0;i<N;i++)
    {
        *(n+i) = i;
        pthread_create( (tid+i),NULL,diner,(void*)(n+i) );
```

```
    }
    for (i=0;i<N;i++)
    {
        pthread_join( *(tid+i),NULL );
    }
    pthread_exit(0); //...w końcu to też wątek
}
```

```
$ ./dining-1
[1]...hmm...co ja to miałem
[2]...hmm...co ja to miałem
[3]...hmm...co ja to miałem
[4]...hmm...co ja to miałem
[5]...hmm...co ja to miałem
[1]...głodny
[1]...obiad...start
[2]...głodny
[1]...obiad...stop
[2]...obiad...start
[3]...głodny
[4]...głodny
[4]...obiad...start
[5]...głodny
[5]...obiad...start
[2]...obiad...stop
[4]...obiad...stop
[5]...obiad...stop
[3]...obiad...start
[3]...obiad...stop
```

Analiza działania programu:

- Każdy filozof jest reprezentowany przez osobny wątek.
- Filozofowie wykonują się w nieskończonej pętli, z których każda reprezentuje jedno "cykle" - próby zjedzenia posiłku określoną ilość razy (zdefiniowane przez zmienną `CYCLE`).
- Każdy filozof wyświetla komunikaty, mówiące o swoim myśleniu, głodzie i próbach zjedzenia.
- Filozofowie próbują uzyskać dostęp do dwóch pałeczek (reprezentowanych przez semafory) po swoich stronach stołu.
- Jeśli filozof o numerze `v` uzyska dostęp do dwóch pałeczek (blokując je semaforami), to oznacza to, że zaczyna jeść przez określony czas (`sleep(1)`), po czym zwalnia pałeczki (odblokowuje semafory).

Wynik działania programu:

- Filozofowie są identyfikowani numerami od 1 do 5.
- Widzimy, że filozofowie zaczynają od stanu myślenia, przechodzą przez stan głodu i próbują jeść.
- Jednak tylko dwóch filozofów jednocześnie uzyskuje dostęp do pałeczek i zaczyna jeść. Pozostali filozofowie pozostają w stanie głodu i próbują uzyskać dostęp do pałeczek, ale nie są w stanie tego zrobić.

Analiza wyniku:

- Wynik pokazuje, że tylko dwóch filozofów jednocześnie jest w stanie zjeść. Pozostali filozofowie czekają na dostęp do pałeczek, ale z powodu wyścigów i konfliktów w dostępie do zasobów (pałeczek), większość z nich pozostaje w stanie głodu.
- Problemem jest zakleszczenie zasobów - dwa filozofowie o numerach parzystych próbują uzyskać dostęp do tych samych dwóch pałeczek jednocześnie, co prowadzi do sytuacji, w której nikt nie może zjeść.
- Zmiana algorytmu lub podejścia do przypisania dostępu do pałeczek (semaforów) może pomóc w rozwiązaniu problemu zakleszczenia i umożliwić równomierny dostęp filozofów do posiłków.

Jak w powyższych wnioskach, program pomimo tego że działa poprawnie jest dość chaotyczny i istnieje mała szansa że dojdzie do zakleszczenia.

Można byłoby przyjąć mniej ofensywną strategię – czekamy cierpliwie do momentu aż będziemy mogli podnieść obie pałeczki. Jest to rozwiązanie plasujące się, niejako, na przeciwległym biegunie. W przypadku gdy proces (wątek) korzysta równocześnie z wielu współdzielonych zasobów a obierze tego rodzaju strategię może to prowadzić do zagłodzenia (process starvation).

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define N 5
#define CYCLE 1
pthread_mutex_t stick[N];
pthread_mutex_t eat; //...oprócz tego że wprowadzimy tu dodatkowy semafor

void* diner( void* arg )
{
    int v;
    int eating = 0;
    v = *((int*)arg) +1;
    while( eating<CYCLE )
    {
        printf( "[%d]...hmm...co ja to miałem\n",v ); sleep( v/2);
        printf( "[%d]...głodny\n", v);
        pthread_mutex_lock( &eat );
        pthread_mutex_lock( (stick+v) );
        pthread_mutex_lock( (stick + (v+1)%N) );
        pthread_mutex_unlock( &eat );
        printf( "[%d]...obiad...start\n",v );
        eating++; sleep(1);
        printf( "[%d]...obiad...stop\n",v );
        pthread_mutex_unlock( (stick+v) );
        pthread_mutex_unlock( (stick + (v+1)%N ) );
    }
    pthread_exit(NULL);
}

int main( void )
{
```

```
    pthread_t tid[N];
    int n[N],i;
    pthread_mutex_init( &eat,NULL );
    for( i=0;i<N;i++ )
    {
        pthread_mutex_init( (stick+i), NULL );
    }
    for( i=0;i<N;i++ )
    {
        *(n+i) = i; pthread_create( (tid+i),NULL,diner,(void*)(n+i) );
    }
    for( i=0;i<N;i++ )
    {
        pthread_join(tid[i],NULL);
    }
    pthread_exit(0);
}
```

```
$ ./dining-2
[1]...hmm...co ja to miałem
[2]...hmm...co ja to miałem
[4]...hmm...co ja to miałem
[3]...hmm...co ja to miałem
[5]...hmm...co ja to miałem
[1]...głodny
[1]...obiad...start
[2]...głodny
[3]...głodny
[1]...obiad...stop
[2]...obiad...start
[5]...głodny
[4]...głodny
[2]...obiad...stop
[3]...obiad...start
[5]...obiad...start
[3]...obiad...stop
[4]...obiad...start
[5]...obiad...stop
[4]...obiad...stop
```

Wprowadzone zmiany w programie uwzględniają dodatkowy semafor (`eat`), który został użyty do koordynacji dostępu do zasobów (pałeczek). Wszystkie semafory, zarówno `stick` jak i `eat`, zostały zainicjowane jako binarne semafory w celu synchronizacji dostępu do zasobów.

Analiza działania programu po zmianach:

- Program wykonuje się w ten sam sposób jak poprzednio, z tą różnicą, że został wprowadzony dodatkowy semafor `eat`, którego blokada jest wykonywana na początku pętli przez filozofa.
- Semafor `eat` jest wykorzystywany jako zabezpieczenie, aby tylko jeden filozof mógł próbować uzyskać dostęp do pałeczek jednocześnie. W miarę jak filozofowie są głodni, próbują uzyskać dostęp do dwóch

pałeczek.
- Filozofowie są w stanie uzyskać dostęp do pałeczek w sposób bardziej równomierny niż w poprzedniej wersji programu.
- Dodanie semafora `eat` pomaga w uniknięciu sytuacji, w której więcej niż jedno "trio" filozofów jednocześnie próbuje uzyskać dostęp do pałeczek.
- Wynik programu pokazuje, że wszystkie pięć wątków/filozofów ma szansę zjeść swoje posiłki, choć niekoniecznie w pełni równolegle.

Wnioski:

- Dodanie dodatkowego semafora (`eat`) pomogło w poprawie równoważenia dostępu do zasobów (pałeczek) przez filozofów.
- Pomimo wprowadzenia zmian, wciąż mogą wystąpić sytuacje, w których niektórzy filozofowie będą czekać na dostęp do pałeczek, szczególnie w przypadku, gdy więcej niż jeden filozof jednocześnie jest głodny.
- Nadal jest to rozwiązanie uproszczone, które może prowadzić do zakleszczenia (szczególnie w bardziej złożonych scenariuszach), jednak w tym konkretnym przypadku wprowadzone zmiany poprawiają równoważenie dostępu do zasobów i zmniejszają ryzyko, że wszyscy filozofowie będą czekać w nieskończoność na posiłek.

Wnioski dla poszczególnych programów

1. **Problem producenta i konsumenta:**
    - Pierwszy kod nie gwarantuje poprawności i bezpieczeństwa dostępu do współdzielonych danych.
    - Drugi kod poprawia ten problem poprzez zastosowanie dodatkowego semafora (`delay`) do synchronizacji producenta i konsumenta.
    - Drugi kod rozwiązuje problem poprzez wykorzystanie dwóch semaforów (jeden do opóźnienia operacji konsumenta, drugi do wykluczania zapisu przez producenta), co eliminuje błędy synchronizacji w poprzednim kodzie.
2. **Problem czytelników i pisarzy:**
    - W pierwotnym kodzie, wyścigi pomiędzy czytelnikami i pisarzami prowadzą do zakleszczeń i braku równomiernego dostępu do zasobów.
    - Problemem jest niewłaściwe zarządzanie zasobami i brak koordynacji, co prowadzi do problemów synchronizacyjnych.
    - Konieczne jest zastosowanie odpowiedniej strategii dostępu do zasobów, aby uniknąć zakleszczeń i zapewnić równomierny dostęp czytelników i pisarzy.
3. **Problem filozofów przy stole:**
    - Pierwotny kod działa, ale może prowadzić do problemów synchronizacyjnych (zakleszczeń) w bardziej złożonych scenariuszach.
    - Wprowadzenie dodatkowego semafora (`eat`) poprawia równowagę dostępu do zasobów przez filozofów, zmniejszając ryzyko zakleszczeń i umożliwiając lepsze wykorzystanie zasobów.
    - Nadal istnieje ryzyko, że niektórzy filozofowie mogą czekać na dostęp do zasobów, ale zmiany pomogły w bardziej równomiernym dostępie do pałeczek.

Wszystkie omówione programy pokazują różne podejścia i strategie do rozwiązywania problemów synchronizacyjnych w wielowątkowym środowisku. Poprawne zarządzanie zasobami i zastosowanie odpowiednich mechanizmów synchronizacyjnych jest kluczowe dla uniknięcia zakleszczeń i zapewnienia poprawnego działania programów wielowątkowych.