WYBRANE PROBLEMY I ZADANIA

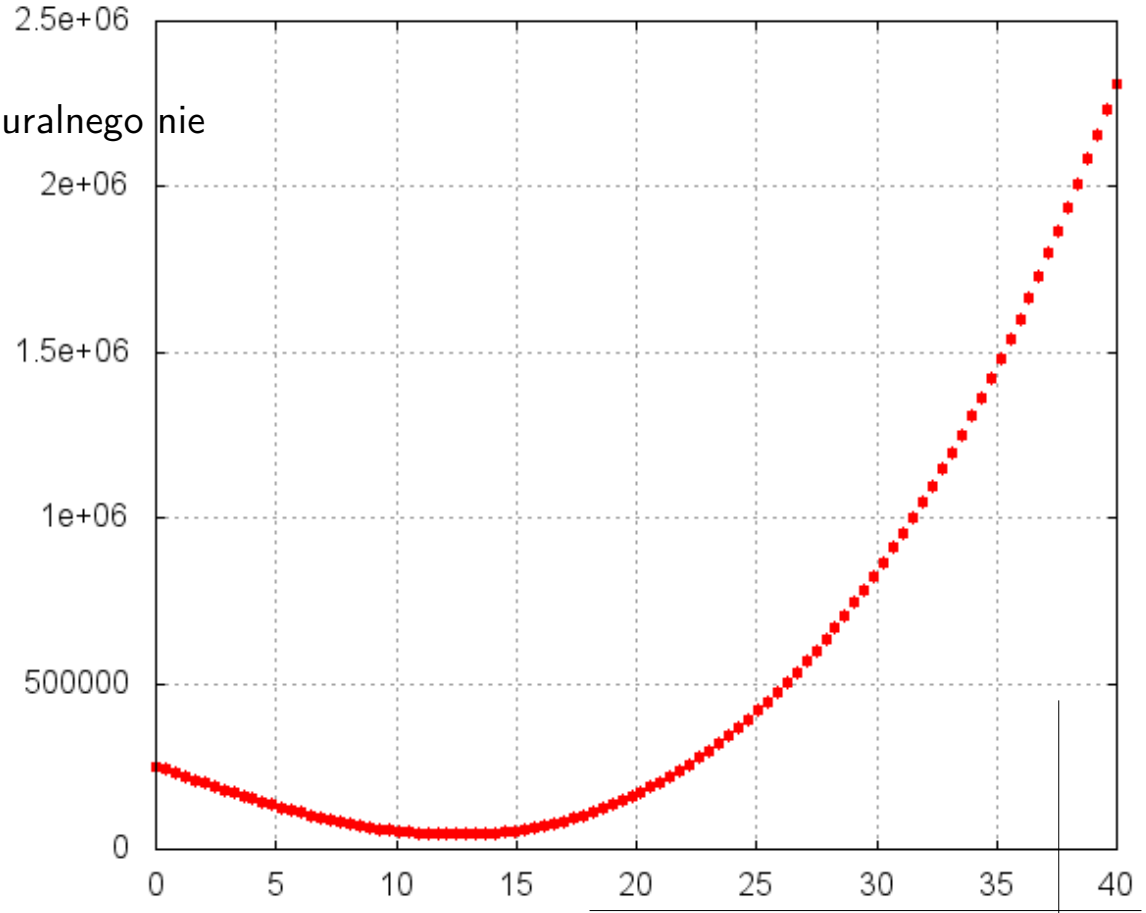
Rozpatrzmy na początek problem typu **`producer`**/**`consumer`**.

Producent będzie generował liczby pierwsze, z użyciem formuły wielomianowej *Jarosława Wróblewskiego* i *Jean-Charles'a Meyrignac'a* (2006), ogólnej postaci

$$42 \cdot n^3 + 270 \cdot n^2 - 2643 \cdot n + 250703$$

dająca wartości liczb pierwszych dla n naturalnego nie większego od 40.

Oczywiście, konsument będzie pobierał i wykorzystywał te wartości.



WYBRANE PROBLEMY I ZADANIA

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

❶ program pobierze z linii komend argument określający ilość liczb pierwszych jaka będzie wygenerowana

❷ producent i konsument stanowić będą dwa wątki o funkcjach **`producer()`** i **`consumer()`**, odpowiednio

❸ wymiana danych między producentem a konsumentem zachodzić będzie przez bufor globalny **`buffer`** (tutaj jedno-elementowy);

❹ z pewnością dostęp do bufora powinien być operacją wyłączną, w takim razie wprowadzimy smafor binarny mutex.

```
#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
```

WYBRANE PROBLEMY I ZADANIA

Potrzebne będą trzy funkcje służące działaniu na buforze ze strony producenta i konsumenta.
Producenta, generująca kolejną liczbą pierwszą

```
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);
}
```

Dodająca nową wartość do bufora, używana przez producenta

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

Służąca pobieraniu wartości z bufora przez konsumenta

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

Oczywiście te funkcje przedstawiają się w tym momencie bardzo elementarnie, jako że przykład ten należy traktować jako swoisty schemat/szablon do dalszej rozbudowy.

WYBRANE PROBLEMY I ZADANIA

Dalej mamy funkcje wątku producenta i konsumenta

```
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);
}
```

WYBRANE PROBLEMY I ZADANIA

Wątek konsumenta (zachowując maksymalną ogólność)

```
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);
}
```

WYBRANE PROBLEMY I ZADANIA

Złóżmy to wszystko w jedną całość (z linii komend pobieramy ilość cykli – tu: wygenerowanych liczb).

```
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;
}
```

No i – w zasadzie – wszystko wydaje się być bardzo proste.

WYBRANE PROBLEMY I ZADANIA

Przyjrzyjmy się w takim razie wykonaniu programu, powiedzmy że dla 10 cykli

```
$ ./pc-1 10
[1082132800] consumer, start
0 -> 0
1 -> 0
2 -> 0
3 -> 0
4 -> 0
5 -> 0
6 -> 0
7 -> 0
8 -> 0
9 -> 0
[1082132800] consumer, stop
```

Zatem wynik jest zupełnie nieadekwatny do oczekiwań – całkowicie brak koordynacji między producentem a konsumentem.

W takim razie kod wymaga uzupełnień w tym zakresie.

WYBRANE PROBLEMY I ZADANIA

Chcąc osiągnąć koordynację między producentem a konsumentem potrzebne będą dwa semafory mutex (ewentualnie jeden i dodatkowo jakaś globalna zmienna sterująca sekwencyjnością).

Zastosujemy tutaj wariant z dwoma semaforami, wprowadzając drobne modyfikacje kodu.

```
#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);
}
```

WYBRANE PROBLEMY I ZADANIA

Producent, kiedy będzie chciał wykonać operację na buforze zgłosi żądanie wyłączności, wykonując **wait()** na semaforze własnym (tu: **exclude**). Natomiast po zakończeniu tej operacji zwolni (ewentualnie) oczekującego konsumenta wykonując **signal()** na jego semaforze (tu: **delay**).

```
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);
}
```

WYBRANE PROBLEMY I ZADANIA

Nie jako na zasadzie symetrii u konsumenta wprowadzimy

```
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);
}
```

Modyfikacja ciała funkcji **main()** nasuwają się już same przez się.

WYBRANE PROBLEMY I ZADANIA

```
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;
}
```

Efekt wykonania, załóżmy 5 cykli

```
./pc-3 5
[1082132800] consumer, start
[1090525504] producer, start
0 -> 250703
1 -> 224579
2 -> 199247
3 -> 174959
4 -> 151967
[1082132800] consumer, stop
```

WYBRANE PROBLEMY I ZADANIA

Zajmijmy się teraz kolejnym klasycznym problemem współbieżności **readers/writers**.
Załóżmy na wstępie:

❶ będziemy mieli R czytelników i W pisarzy a każdy z nich dokona N cykli zapisu, po czym program zakończy działanie;

❷ jest rzeczą naturalną, że wykorzystamy tu dwa semafory (w zupełności wystarczającymi będą binarne mutex);

❸ sformułowanie rozwiązania może być bardzo różne, zależnie od zamierzonego efektu – tutaj przyjmiemy preferencję dla czytelników.

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

int counter;   //...zmienna monitorująca ilość czytelników w czytelni
```

WYBRANE PROBLEMY I ZADANIA

Funkcja wątku pisarza przedstawia się następująco.

```
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 );
}
```

WYBRANE PROBLEMY I ZADANIA

Kod wątku dla czytelników.

```
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);
}
```

WYBRANE PROBLEMY I ZADANIA

No koniec ciało funkcji **main()**.

```
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;
}
```

```
$ ./rw
[1115703616] pisarz, start
[1115703616] pisarz, stop
[1115703616] pisarz, start
[1115703616] pisarz, stop
[1082132800] czytelnik, start
[1090525504] czytelnik, start
[1098918208] czytelnik, start
[1107310912] czytelnik, start
[1082132800] czytelnik, stop
[1082132800] czytelnik, start
[1090525504] czytelnik, stop
[1090525504] czytelnik, start
[1098918208] czytelnik, stop
[1098918208] czytelnik, start
[1107310912] czytelnik, stop
[1107310912] czytelnik, start
[1124096320] pisarz, stop
[1124096320] pisarz, start
[1124096320] pisarz, stop
[1115703616] pisarz, start
[1115703616] pisarz, stop
```

itd., itd., itd.

WYBRANE PROBLEMY I ZADANIA

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

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
}
```

WYBRANE PROBLEMY I ZADANIA

Czas na funkcję wątku **diner()**.

```
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);
}
```

Efekt wykonania (zakładając że program skompilowano pod nazwą **dining-1**)

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

WYBRANE PROBLEMY I ZADANIA

Przedstawione rozwiązanie zakłada strategię działania polegającą na natychmiastowym przechwyceniu własnej pałeczki

```
pthread_mutex_lock( (sticks+v) );
```

i następującej po niej próbie przechwycenia pałeczki sąsiada

```
pthread_mutex_lock( (sticks + (v+1)%N ) );
```

Jest to rozwiązanie proste zazwyczaj skuteczne, nie wyklucza jednak możliwości wystąpienia sytuacji ***zakleszczenia*** (***deadlock***) – co stanie się wówczas jeżeli wszyscy próbujący uzyskać dostęp do zasobu postąpią identycznie w tym samym momencie. Praktycznie jest to małorealne, aczkolwiek nie można tego całkowicie wykluczyć.

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***).

Spróbujmy zrealizować teraz ten drugi wariant.

WYBRANE PROBLEMY I ZADANIA

W części początkowej programu wymagane zmiany raczej niewielkie.

```
#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

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);
}
```

WYBRANE PROBLEMY I ZADANIA

Funkcja wątku, jak niżej.

```
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);
}
```

WYBRANE PROBLEMY I ZADANIA

Efekt wykonania

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

WYBRANE PROBLEMY I ZADANIA