UŻYCIE I ZARZĄDZANIE WĄTKAMI

Proces jako pewna instancja programu, w trakcie wykonania, ze swej natury w każdym systemie operacyjnym wyróżniają:

- prawa własności zasobu a jednym z fundamentalnych zadań systemu jest ochrona przed jednoczesnym dostępem;
- szeregowanie i wykonanie procesów odbywa się z poziomu systemu operacyjnego .

W odróżnieniu od procesu, wątek stanowi podzbiór przestrzeni adresowej procesu, współdzielący jego stan i zasoby (również deskryptory plików) – aczkolwiek nie dziedziczy stosu procesu (***stack***).

Dzięki temu komunikacja między wątkowa nie wymaga użycia systemowych mechanizmów ***inter-process communication*** a przełączanie kontekstu (***context switch***) jest niewspółmiernie szybsze niż w przypadku procesu.

Z racji wygody użycia istnieje wiele implementacji bibliotek obsługujących wielowątkowość.
Dla systemów ***POSIX*** standardem jest ***POSIX 1003.1c standard (1995)***
pthreads, czyli ***POSIX THREADS***
Ponieważ jest uzupełnieniem API systemowego, wymagana jest jawna konsolidacja z `libpthread.so` (albo `libpthread.a`)

```
gcc -Wall <source>.c -o <exec> -lpthread
```

O wątkach ***LINUX***owych więcej
http://sunsite.unc.edu/pub/Linux/docs/faqs/Threads-FAQ/html/

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Podobnie jak każdemu procesowi w chwili jego tworzenia przypisywane jest unikalne

```
pid_t id;
```

tak też i wątkowi przypisywane jest

```
pthread_t id;
```

Wątek może pobrać swój własny identyfikator wywołaniem

```
#include <pthread.h>
pthrad_t pthread_self(void);
```

Tworzenie nowego, wątku potomnego odbywa się wywołaniem funkcji **pthread_create()**

SYNOPSIS

```
#include <pthread.h>
int pthread_create( pthread_t *tid,const pthread_attr_t *attr,
    void *(*thread)(void*),void *arg );
pthread_t *tid, identyfikator nowotworzonego wątku (jeżeli sukces)
pthread_attr_t*attr,sposób dołączania,kolejkowania wątku (domyślnie NULL)
thread(), funkcja wątku
void *arg, argument dla funkcji startowej, NULL jeżeli brak
```

RETURN

0 ,*jeżeli sukces*

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Zakończenie wątku może nastąpić z czterech przyczyn:

- zwrócenie sterowania z wątku (wywołanie **return**, **exit()**, **_exit()**);
- wywołanie z wątku nadrzędnego (macierzystego);
- z innego wątku wywołaniem

```
#include <pthread.h>
int pthread_cancel( pthread_t tid );
```

- jawne wywołanie funkcji **pthread_exit()** z kodem powrotu **code**.

```
#include <pthread.h>
void pthread_exit( void *code );
```

Wykonajmy najpierw prosty przykład kiedy dwa wątki będą pisać na konsoli.
Kod ciała wątku głównego przedstawiał się będzie następująco

```
#include<stdio.h>
#include<pthread.h>
#define ENDLESS 1

int main( void )
{
    pthread_t tid;
    pthread_create( &tid,NULL,&o,NULL );
    while( ENDLESS ){ putchar( 'x' ); }
    return 0;
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Funkcja wątku będzie miała prostą definicję

```
void* o( void* unused )
{
    (void)unused;
    while( ENDLESS ){ putchar( 'o' ); }
    return 0;
}
```

Jeżeli zapiszemy kod źródłowy pod nazwą **xox.c,** to kompilację i konsolidację wykonujemy komendą

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

a wykonanie, o ile powyższa akcja zakończyła się sukcesem

```
$ ./xox
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xooooooooooooooooooooooooooooooooooooo^C
```

Program ten, a ściślej wątki możemy zatrzymać wyłącznie wysyłając odpowiedni sygnał.

Zwróćmy uwagę, że tak jak wspominano to już wcześniej, nie możemy czynić żadnych założeń a priori odnośnie relacji czasowych wykonujących się równolegle zadań.

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Przygotujemy teraz program, którego proces utworzy tyle wątków ile podanych będzie w linii wywołania programu.

```
Zaczynamy od deklaracji plików nagłówkowych
#include <stdlib.h> ... bo exit() i stałe symboliczne
#include <stdio.h>  ... operacje wejścia-wyjścia
#include <unistd.h> ... pobranie PID procesu, przy pomocy getpid()
#include <pthread.h>... wątki, oczywiście niezbędne

Deklaracja nagłówkowa funkcji main()
int main( int argc,char *argv[] )
{
   pthread_t tid;   ... tu będzie zwracane TID  przez pthread_create()
   int rc;          ... wartość zwracana przez pthread_create()
   long i,n;        ... zmienne sterujące pętli tworzącej wątki

   Deklaracja funkcji wątku.
   void* hello( void*);

   ... i możemy przystępować do rzeczy
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

```
    Sprawdzamy, czy aby wywołanie programu było poprawne
    if( argc>1 ) ... no można by właściwie arrc==2
    {
        Najpierw czytamy z linii wywołania ilość wątków, do zmiennej n
        sscanf( argv[1],"%ld",&n );
        Przystępujemy do tworzenia zadanej ilość wątków
        for( i=0;i<n;i++ )
        {
            Informacja diagnostyczna
            printf( "PID[%ld] tworzy wątek,...#%ld...\n",
                (long)getpid(),(i+1));
            Tworzymy nowy wątek - (i+1) żaby pierwszy był 1
            rc = pthread_create( &tid, NULL,hello,(void *)(i+1));
            Drobna diagnostyka błędów
            if(rc){perror( "!.!.!...błąd pthread_create()...");exit(rc);}
        }
    }
    else ... na wypadek błędnego wywołania programu
    { printf( "!.!.!... wywołanie powinno mieć postać: %s %s\n",
            argv[0],"<ilość_wątków>" ); }
    Oczywiście proces można zakończyć i tak ...
   pthread_exit( NULL );   ... wszakże jest wątkiem !
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Pozostało jeszcze przygotować funkcję wątku **hello()**.

```
void* hello( void *n )
{
    Wyświetlamy informację diagnostyczną
    printf("PID[%ld] ...jestem wątkiem #%ld!  TID[%d]\n",
       (long)getpid(),(long)n,(int)pthread_self() );
    i wątek kończy działanie
    pthread_exit( NULL ); ...właściwie to wywołanie jest zbędne
}
```

Kompilacja i konsolidacja – jeżeli kod źródłowy zapisano pod nazwą **hello.c** - oczywiście

```
$ gcc -Wall helllo.c -o hello -lpthread
```

Uruchomienie

```
$ ./hello 4
PID[5874] tworzy wątek,...          #1...
PID[5874] tworzy wątek,...          #2...
PID[5874] tworzy wątek,...          #3...
PID[5874] ...jestem wątkiem #1!  TID[1082132800]
PID[5874] ...jestem wątkiem #2!  TID[1090525504]
PID[5874] ...jestem wątkiem #3!  TID[1098918208]
PID[5874] tworzy wątek,...          #4...
PID[5874] ...jestem wątkiem #4!  TID[1107310912]
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Specyfikacja parametru formalnego funkcji wątku typu `void*` może wydawać się dziwna i niewygodna, jest jednak jednym jaka może być zastosowana w tym celu w ramach C strukturalnego.

W kolejnym przykładzie prześlemy, z procesu głównego do wątku potomnego, tablicę liczbową a tam zostanie wyznaczona wartość średnia jej elementów.

```
Funkcja wątku może się przedstawiać w takim razie jak niżej
void* thread( void* array )
{
    int i;
    double sum,avg;
    Obliczamy sumę elementów tablicy,
    zwróćmy uwagę na konwersję typu wskazania (void*) na (double*)
    for( i=0,sum=0.0;i<n;i++ ){ sum += *( (double*)array+i ); }
    Wyznaczamy średnią arytmetyczną
    avg = sum/n;
    i wyprowadzamy na konsolę wynik
    printf( "wartość średnia: %16.10f\n",sum/(double)n );

    return ( (void*)0 );
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Natomiast kod procesu głównego

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define n 10000 ...rozmiar tablicy - cóż można by też jak wcześniej
int main( void )
{
    int i;
    pthread_t tid;
    double x[n];  ... można też dynamicznie
    // double *x;
    // x = (double*) calloc( size_t n, sizeof( double ) );
    // free( (void*)x );

    for( i=0;i<n;i++ ){ *(x+i) = ((double)rand())/(RAND_MAX); }
    printf( "wysyłam dane do wątku...[%d]\n",(int)tid );
    pthread_create( &tid, NULL,thread,(void *)x );
    printf( "...czekam na wątek\n" );

    return 0;
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

No i teraz wykonanie (powiedzmy że skompilowano pod nazwą **table**)

```
$ ./table
wysyłam dane do wątku...[0]
...czekam na wątek
```

a wynik niezbyt budujący – proces nadrzędny zakończył zanim wątek wykonał obliczenia.

Można byłoby się ratować - przykładowo - użyciem **sleep()**,
czyli zmodyfikujmy kod **main()**, dodając po **printf()**

```
printf( "...czekam na wątek\n" );
sleep( 1 );
```

co wymusi oczekiwanie przez 1 sekundę. A efekt

```
$ ./table
wysyłam dane do wątku...[73419]
...czekam na wątek
wartość średnia:     0.4971322194
```

poprawny – ponieważ tablicę zainicjowaliśmy z generatora zmiennych pseudolosowych **rand()**, normalizując od **0.0** do **1.0**, to wartość średnia powinna wynieść dokładnie

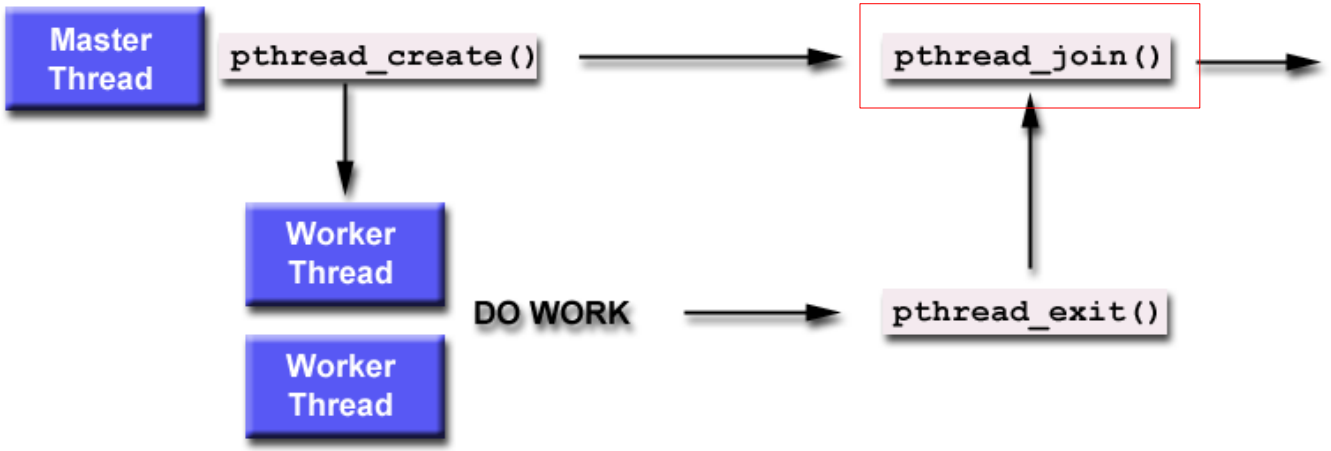
$$\overline{x} = \frac{1}{2}$$

Aczkolwiek sposób rozwiązania problemu nieco sztuczny i nienaturalny.

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Jeżeli dzielimy większy problem na zadania i przekazujemy ich wykonanie wątkom, to powinniśmy zagwarantować że proces nadrzędny zaczeka aż wątek zakończy, w przeciwnym przypadku całość nie ma większego sensu. Stosowanie rozwiązań w rodzaju **`sleep()`** jest bardzo nieefektywne a bywa że i nie skuteczne.

Służy temu m.in. funkcja **`phread_join()`**.



SYNOPSIS

```
#include <pthread.h>
int pthread_join( pthread_t tid, void **code );
pthread_t tid identyfikator wątka, na który należy czekać
void **code kod powrotu wątka (wartość z pthread_exit())
```

RETURN

```
0 ,jeżeli sukces (kod błędu w przeciwnym razie)
```

Funkcja zawiesza, w razie potrzeby, wykonanie procesu (wątku) wywołującego **`pthread_join()`**, aż do momentu kiedy podany wątek **`tid`** zakończy działanie.

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Teraz elementarna demonstracja użycia funkcji **pthread_join()**.

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

int main( void )
{
    pthread_t tid;
    int rc;
    void* thread( void* );
    Tworzymy nowy wątek
    rc = pthread_create( &tid, NULL,thread,NULL );
    if( rc ){ perror( "!.!.!...pthread_create()..." ); exit( 1 ); }
    else
    {
        if( pthread_join ( tid, NULL ) ) ... i próbujemy połączyć
        { perror( "!.!.!...pthread_join()..."  ); exit( 2 ); }
    }

  return 0;
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Funkcja wątku – po prostu – wymusi odczekanie 5 sekund ma procesie nadrzędnym.

```
void *thread( void *arg )
{
    int i;
    printf( "...w wątku\n" ); fflush( stdout );
    for ( i=0;i<5;i++ )
    { printf("\t%3d s\n",(i+1) ); fflush( stdout ); sleep(1); }
    printf( "...i już koniec, zwracam sterowanie\n" ); fflush( stdout );

    pthread_exit( NULL );
}
```

Wykonanie

```
$ ./join
...w wątku
    1 s
    2 s
    3 s
    4 s
    5 s
...i już koniec, zwracam sterowanie
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Funkcja **pthread_join()**, ze swej natury, służy przyłączeniu jednego wątku co może stanowić pewnie problem jeżeli wątków będzie więcej.

Załóżmy, że funkcja wątku jest następująca

```
void *thread( void *n )
{
    int i;
    double sum;
    printf("...wątek %3ld startuje...\n",(long)n ); fflush( stdout );
    for( i=0,sum=0.0;i<N;i++ )
    {
        sum +=    sin((double)i)*sin((double)i) +
                  cos((double)i)*cos((double)i) - 1.0;
    }
    printf("...wątek %3ld zakończył...suma = %e\n",(long)n,sum );
    fflush( stdout );
    pthread_exit( NULL );
}
```

Zauważmy że wartością dokładną sumy jest zero, ponieważ zawsze

$$sin(\ \alpha\ )^2 + cos(\ \alpha\ )^2\ =\ 1$$

UŻYCIE I ZARZĄDZANIE WĄTKAMI

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

#define T 10
#define N 100000

int main( void )
{
   pthread_t threads[T];
   int rc;
   long t;

    Uruchamiamy wątki
    for( t=0;t<T;t++)
    {
        rc = pthread_create( &threads[t],NULL,thread,(void *)(t+1) );
        if (rc) { perror( "!.!.!...pthread_create()..." ); exit( 1 ); }
    }

    ... i zbieramy ponownie w całość
    for( t=0;t<T;t++)
    {
        rc = pthread_join( threads[t],NULL );
        if (rc){ perror( "!.!.!...pthread_join()..." ); exit( 2 ); }
    }

    pthread_exit(NULL);
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Jednym z kluczowych względów zastosowania wątków na gruncie współbieżności, prócz wymienionych na wstępnie, są kwestie efektywności.

Cele oszacowania czasu wykonywania można wykorzystać komendę systemową

```
time <command>
```

podaje ona oszacowanie czasu dla procesu, z podziałem na składowe

- **real**, w przybliżeniu jest to

$$\texttt{real} \approx \texttt{user} + \texttt{sys}$$

jednak w przypadku systemów multi-tasking, dodatkowo pracujących przy dużym obciążeniu

$$\texttt{real} \gg \texttt{user} + \texttt{sys}$$

- **user**, jest to czas kiedy *CPU* pozostawał w trybie user mode, a więc wykonywane były zakodowane instrukcje czy były wywołanie funkcji bibliotecznych typu **printf()**, **malloc()**, **strlen()**, **fopen()**, i podobnych;
- **sys** , jest to czas kiedy *CPU* pozostawał w trybie kernel mode, a więc wystąpiły wywołania funkcji systemowych w rodzaju **open()**, **read()**, **write()**, **close()**, **wait()**, **exec()**, **fork()**, **exit()**, i podobnych.

Przykładowo:

```
$ time sleep 5
real    0m5.006s
user    0m0.000s
sys     0m0.004s
```

czyli uśpienie na 5 sekund zajęło de facto 5.006 sekundy, z czego na wywołania funkcji użytkownika wypadło 0 (bo ich nie było) a systemowych 4 milisekundy. Zauważmy, że pojawiły się też 2 milisekundy opóźnienia wynikła z faktu, że nie jest to w końcu jedyny proces obsługiwany przez system

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Załóżmy że wykonamy określoną ilość razy funkcję. W celach testowania założymy, że będzie to funkcja, która wykonuje dodawanie

$$1 + 1$$

i podstawia pod zmienną rzeczywistą.

Kod tego rodzaju funkcji może mieć postać

```
void* task( void* arg )
{
    double x;
    x = 1.0 + 1.0;
    return NULL;
}
```

Celem porównania efektywności wątków i procesów funkcja **task()** wywołana będzie z dla zadanej ilości:

- procesów, będzie ona wykonywana przez potomka powołanego wywołaniem **fork()**;
- wątków, będzie to po prostu funkcja wątku, zadana w wywołaniu **pthread_create()**;

UŻYCIE I ZARZĄDZANIE WĄTKAMI

W przypadku programu posługującego się procesami, kod przedstawiał się będzie następująco.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <limits.h>

int main( int argc,char** argv )
{
   unsigned long i,n;
   int status;
   pid_t pid;
   if( argc>1 )
   {
      if( sscanf( argv[1],"%lu",&n )==1 )
      {
         for( i=0;i<n;i++ )
         {
            switch( (int)(pid=fork()) )
            {
               case -1: perror( "!.!.!...fork()..." ); exit( 1 ); break;
               case  0: task( NULL ); exit( 0 ); break;
               default: waitpid( pid,&status,0 );
            }
         }
       }else{ printf( "!.!.!...błędny argument [%s] wywołania [%s]\n",argv[1],argv[0] );  }
   }else{ printf( "!.!.!... %s [ilość < ULONG_MAX=%lu]\n",argv[0],ULONG_MAX );  }
   return 0;
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

A teraz kod dla wątków

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

int main( int argc,char** argv )
{
   unsigned long i,n;
   pthread_t tid;
   if( argc>1 )
   {
      if( sscanf( argv[1],"%lu",&n )==1 )
      {
         for( i=0;i<n;i++ )
         {
            if( pthread_create( &tid, NULL,task,NULL ) )
            { perror( "!.!.!...fork()..." ); exit( 1 ); }
            else
            { pthread_join ( tid, NULL ); }
         }
       }else{ printf( "!.!.!...błędny argument [%s] wywołania [%s]\n",argv[1],argv[0] );  }
   }
   else{ printf( "!.!.!... %s [ilość < ULONG_MAX=%lu]\n",argv[0],ULONG_MAX );  }

   return 0;
}
```

UŻYCIE I ZARZĄDZANIE WĄTKAMI

Prezentowane dalej wyniki dotyczą procesora

`AMD Turion(tm) 64 Mobile Technology MK-38 (2.2 GHz, BOGOMIPS: 1597.66)`

pracującego pod kontrolą

`Linux, kernel 2.6.18`

Kompilacje wykonano przy użyciu

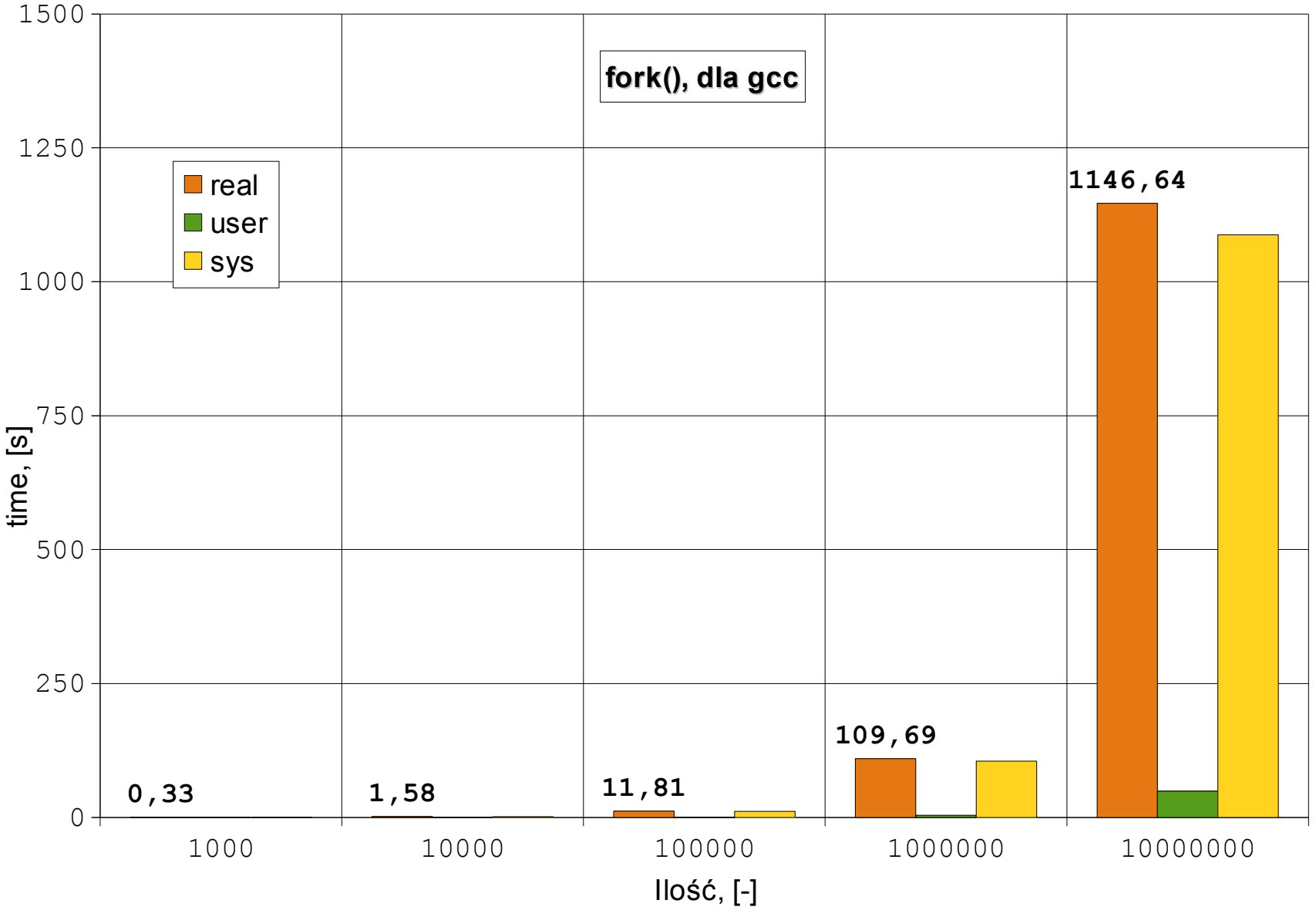
- GNU C compiler (**`gcc`**) ver 4.1.2
- Intel C Compiler (**`icc`**) ver 10.1
- Open64 Compiler Suite (**`opencc`**) ver 4.1

w każdym przypadku z opcją pełnej optymalizacji, czyli

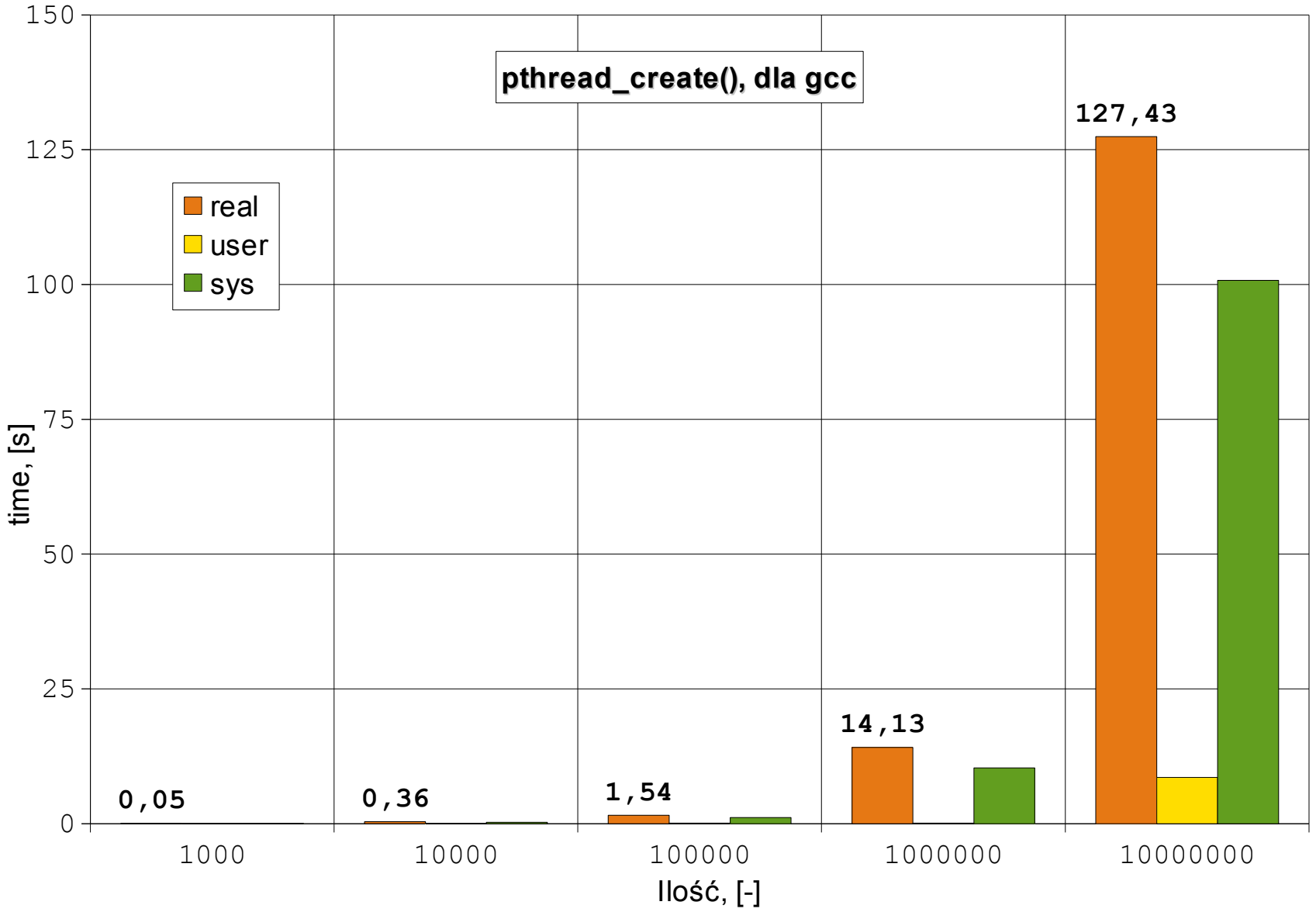
`-O3`

...o jednak tutaj wiele nie wniesie.

UŻYCIE I ZARZĄDZANIE WĄTKAMI



UŻYCIE I ZARZĄDZANIE WĄTKAMI



UŻYCIE I ZARZĄDZANIE WĄTKAMI

Interesujące może być na ile efektywnie zarządza procesami i wątkami wynikowy kod trzech testowanych tu kompilatorów.