

Marek, Kubicki

Programowanie równoległe.

Sprawozdanie z laboratorium 3,4,5.

1. Celem tych zajęć było zapoznanie się z działaniem funkcji z rodziny pthread, a także poszerzenie zrozumienia organizacji pracy wątków.

Rodzina **pthread** daje nam możliwości na inicjalizowanie wątków za pomocą:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,  
void * (*start_routine)(void *), void * arg)
```

funkcja ta tworzy wątek którego **tid** przypisany jest pod zmianą **pthread_t** podaną za pomocą wskaźnika do pierwszego argumentu, z atrybutami wątku podanymi w drugim argumentem. Wątek ten wykona funkcję podaną w argumentem trzecim z argumentami zadanymi w argumentem czwartym.

Wątki stworzone w ten sposób mogą działać na dwa sposoby, w trybie połączonym i odłączonym. Domyślnie wątki startują w trybie połączonym (chyba że zostało to zmienione poprzez zmianę odpowiedniego atrybutu poprzez drugi argument **pthread_create()**), mogą one zostać zmienione w wątki działające w trybie odłączonym za pomocą **pthread_detach()**.

Wątki w trybie połączonym i odłączonym różnią się od siebie głównie w tym kiedy zwalniają zasoby i w sposobie komunikacji z rodzicem. Wątki w trybie połączonym wymagają wywołania **pthread_join()** w celu pełnego zwolnienia zasobów i dodatkowej komunikacji z rodzicem. Wątki w trybie odłączonym zwalniają swoje zasoby po zakończeniu pracy i nie łączą się z wąkiem głównym za pomocą **pthread_join()**, komunikacja z nimi polega na wykorzystywaniu zmiennych globalnych lub wskaźników.

Działanie wątku może być przedwcześnie zakończone za pomocą **pthread_cancel()** i **pthread_kill()**. **pthread_cancel()** działa różnie w zależności od ustawionych flag, jeżeli wątek ustawia flagę **pthread_setcancelstate** na **disable** to prośba o zakończenie działania wątku będzie tymczasowo zignorowana, do czasu ustawienia flagi na **enable**. Podobnie działa **pthread_setcanceltype**, jeżeli ustawiony na **asynchronous** to wątek może przerwać swoje działanie w którymkolwiek momencie działania, jeżeli jednak flaga ta ustawiona jest na **deferred** to wątek zakończy swoje działanie tylko po wywołaniu funkcji która jest **punktem anulowania(cancelation point)**.

Po próbie zakończenia działania pracy wątku można sprawdzić czy zakończyła się ona sukcesem wywołując na nim **pthread_join()** które powinno zwrócić **PTHREAD_CANCELED** jeżeli anulowanie wątku powiodło się, w przypadku wątków w trybie odłączonym należy użyć zmiennych globalnych, lub wskaźników.

Podczas działania na wątkach należy ostrożnie zarządzać dostępem do zasobów krytycznych w celu zapobiegnięcia niepoprawnego zapisu lub odczytu danych, pomaga nam w tym **mutex**. Po stworzeniu **mutexa** za pomocą **pthread_mutex_init()**, można używać go do zablokowania sekcji krytycznych kodu podczas ich wykorzystywania, aby to zrobić należy wywołać **pthread_mutex_lock()** lub **pthread_mutex_trylock()**. Obie funkcje zablokują **mutex** jeżeli był on odblokowany i zwrócią wartość 0. Różnica w ich działaniu polega kiedy zostaną one wywołane na zablokowanym **mutexie**. **pthread_mutex_lock()** będzie oczekiwał na odblokowanie **mutexa**, ale

`pthread_mutex_trylock()` zwróci **EBUSY** pozwalając na wykorzystanie czasu spędzonego na oczekивание на въявление до секции критичной poprzez wykonanie części niekrytycznych operacji.

laboratorium 3

Na laboratorium 3 zadaniem było stworzenie który przy tworzeniu wątków wysyła im unikalny identyfikator, liczby 0 – (n-1), a następnie dla każdego wątku wypisuje wartość tego identyfikatora oraz wynik wywołania `pthread_self()`. W moim przypadku poprawnie działający program używa tablicy z zapisanymi identyfikatorami wysyłanymi poprzez wskaźniki. Funkcja zadanie_watku:

```
void * zadanie_watku (void * arg_wsk)
{
    unsigned int b = *(int*)arg_wsk;
    printf("\twatek potomny: self \nID: %d\n self ID: %lld\n", b, pthread_self());
    return(NULL);
}
```

Pętla tworzenia wątków wraz z tablicą identyfikatorów w main:

```
int id[5] = {0,1,2,3,4};
for(i=0;i<ilosz; i++)
{
    pthread_create(&(tid[i]), NULL, zadanie_watku, &(id[i]));
}
```

Wynik:

```
m@m-VirtualBox:~/PR_lab/lab_3/zad_2$ ./pthr
          watek potomny: self
ID: 1
  self ID: 138731294033472
          watek potomny: self
ID: 0
  self ID: 138731304519232
          watek potomny: self
ID: 2
  self ID: 138731283547712
          watek potomny: self
ID: 4
  self ID: 138731189175872
          watek potomny: self
ID: 3
  self ID: 138731199661632
```

Wątki wypisują swoje identyfikatory w kolejności różniącej się od tego jak były wywołane, ale wypisują odpowiednie ID (od 0 do 4).

Bardziej naiwne rozwiązanie, wysyłanie wskaźnika do pojedyńczej zmiennej ma szansę na błędny wynik.

Zmodyfikowana pętla w main:

```
int id = 0;
for(i=0;i<ilosc; i++)
{
    pthread_create(&(tid[i]), NULL, zadanie_watku, &(id));
    id++;
}
```

Wynik:

```
m@m-VirtualBox:~/PR_lab/lab_3/zad_2$ ./pthr
        watek potomny: self
ID: 2
    self ID: 127195146094144
        watek potomny: self
ID: 3
    self ID: 127195135608384
        watek potomny: self
ID: 5
    self ID: 127194984613440
        watek potomny: self
ID: 5
    self ID: 127195114636864
        watek potomny: self
ID: 3
    self ID: 127195125122624
```

Wątki nadal wypisują swoje ID w różnej kolejności od ich wywołania pomimo tego że tym razem jest to jedna zmienna. Prawdopodobnie jest to spowodowane wyścigiem do dostępu do konsoli(**race condition**). Występują dwa powtórzenia ID pomimo małej próbki, wskazuje to na poważny problem w przesyłce danych, przewidziany powyżej, zmienna może być zmieniana po wysłaniu wskaźnika do wątku, ale przed odczytaniem odpowiedniej wartości.

W dalszej części laboratorium trzeba było stworzyć program wysyłający do wątku strukturę, która następnie będzie wykorzystana do wysłania wyniku obliczeń.

Przekazywanie wskaźnika struktury do wątku, podobnie jak przekazywanie normalnej zmiennej:

```
for(i=0;i<ilosc; i++){
    pthread_create(&(tid[i]), NULL, zadanie_watku, &(s[i]));
}
```

Struktura i funkcja:

```
struct str
{
    double in[2];
    double out;
};

void * zadanie_watku (void * arg_wsk)
{
    struct str s;
    s.in[0] = (*(struct str*)arg_wsk).in[0];
    s.in[1] = (*(struct str*)arg_wsk).in[1];
    s.out = (s.in[0]+s.in[1])/2.0;
    (*(struct str*)arg_wsk).out = s.out;
    return(NULL);
}
```

Struktura przyjmuje wartości wejściowe w polu **in**, następnie obliczana jest też średnia wartość wartości wejściowych i zapisywana jest w **out**, następnie wyniki obliczeń wypisywane są w pętli w mainie:

```
for(i=0;i<ilosc; i++)
{
    printf("IN: %lf-%lf\t OUT: %lf\n", (s[i]).in[0], (s[i]).in[1],(s[i]).out);
    suma +=s[i].out;
}
printf("\nSuma: %lf\n", suma);
}
```

Wynik:

```
m@m-VirtualBox:~/PR_lab/lab_3/zad_3$ ./pthr
IN: 0.000000-1.000000      OUT: 0.500000
IN: 1.000000-2.000000      OUT: 1.500000
IN: 2.000000-3.000000      OUT: 2.500000
IN: 3.000000-4.000000      OUT: 3.500000
IN: 4.000000-5.000000      OUT: 4.500000

Suma: 12.500000
```

Zgodnie z zamierzonym działaniem, wartości średnie danych wejściowych zapisywane są w **out**, sumowane i wypisywane w mainie.

laboratorium 4

Na laboratorium 4 zadaniem była symulacja pubu z zadaną liczbą klientów, kufli i kranów(na potrzeby przeprowadzonego przeze mnie zakresu zadań istnieje tylko jeden kran). Klienci(wątki) przychodzą do pubu, zamawiają kufle, napełniają je, piją ich zawartość i zwracają kufle. Powtarzają one tą pętle zadaną ilość razy(*ile_musze_wypic*) po czym opuszczają oni pub. W pierwszej wersji programu nie ma żadnych zabezpieczeń przed dwoma wątkami próbującymi zmienić ilość kuflów dostępnych w pubie. Stwarza to możliwość wystąpienie błędów synchronizacji i w efekcie zmiany końcowej ilości kuflów.

Reprezentacja pubu w funkcji main:

```
printf("\nOtwieramy pub (simple)!\n");
printf("\nLiczba wolnych kuflów %d\n", l_kf);

for(i=0;i<l_kl;i++){
    pthread_create(&tab_klient[i], NULL, watek_klient, &tab_klient_id[i]);
}
for(i=0;i<l_kl;i++){
    pthread_join(tab_klient[i], NULL);
}
printf("\nLiczba wolnych kuflów %d\n", l_kf);
if(l_kf != l_kfp)
{
    printf("\nLiczba wolnych kuflów zmieniła się o %d\n", l_kfp - l_kf);
}
printf("\nZamykamy pub!\n");
```

Kod funkcji klienta:

```
void * watek_klient (void * arg_wsk){

    int moj_id = * ((int *)arg_wsk);

    int i, j, kufel, result;
    int ile_musze_wypic = ILE_MUSZE_WYPIC;

    long int wykonana_praca = 0;

    //printf("\nKlient %d, wchodzę do pubu\n", moj_id);

    for(i=0; i<ile_musze_wypic; i++){

        printf("\nKlient %d, wybieram kufel\n", moj_id);
        l_kf = l_kf -1;
        if(l_kf < 0)
        {
            printf("\nKlient %d, pobrał nieistniejący kufel\n", moj_id);
        }
        printf("\nKlient %d, podnosi kufel\n", moj_id);

        printf("\nKlient %d, nalewam z kranu %d\n", moj_id, j);
```

```

usleep(30);

printf("\nKlient %d, pije\n", moj_id);
nanosleep((struct timespec[]){
    {0, 50000000L}, NULL);

printf("\nKlient %d, odkladam kufel\n", moj_id);
_lkf++;

}

printf("\nKlient %d, wychodzę z pubu; wykonana praca %ld\n",
moj_id, wykonana_praca);

return(NULL);
}

```

Program pozwala na zadanie dowolnej liczby kufl i klientów (a także na zmianę liczby powtórzeń pętli w funkcji klienta). Nie ma żadnego zabezpieczenia przed klientem pobierającym kufel kiedy nie ma już żadnych wolnych, sprawdzane jest to poprzez warunek if(_kf < 0) tuż po pobraniu kufla(problem tez zostanie rozwiązyany w dalszych wersjach programu)

Pobranie nieistniejącego kufla:

```

Klient 3, wybieram kufel
Klient 3, pobrał nieistniejący kufel
Klient 3, podnosi kufel

```

Innym potencjalnym problemem jest zmiana ilości kufl, po usuinęciu wszystkich fragmentów kodu spowalniających funkcję klienta w celu zwiększenia szansy wystąpienia tego błędu pozostajemy z:

```

void * watek_klient (void * arg_wsk){

int moj_id = * ((int *)arg_wsk);

int i, j, kufel, result;
int ile_musze_wypic = ILE_MUSZE_WYPIC;

long int wykonana_praca = 0;

//printf("\nKlient %d, wchodzę do pubu\n", moj_id);

for(i=0; i<ile_musze_wypic; i++){ // ile_musze_wypic = 20

```

```

    l_kf = l_kf -1;
    usleep(1);
    l_kf++;

}

//printf("\nKlient %d, wychodzę z pubu; wykonana praca %ld\n",
//   moj_id, wykonana_praca);
//zakomentowane w celu czytelniejszego przedstawienia bledu
return(NULL);
}

```

Wynik:

```
m@m-VirtualBox:~/PR_lab/lab_4/3.0$ ./pub_sym_1

Liczba klientów: 100

Liczba kuflí: 100

Otwieramy pub (simple)!

Liczba wolnych kuflí 100

Liczba wolnych kuflí 95

Liczba wolnych kuflí zmienila się o 5

Zamykamy pub!
```

Po wywołaniu programu z 100 kuflami i klientami udało mi się otrzymać inną liczbę kuflí na początku i końcu działania pubu(**uslep(1)**) zapobiega optymalizacji modyfikacji liczby kuflí).

W dalszej części laboratorium moim zmodyfikowałem program w celu zapobiegnięcia tym błędom.

Pierwszym problemem jest synchronizacja podczas pobierania i oddawania kuflí. Najprostszym rozwiążaniem tego problemu jest dodanie **mutexa** blokującego dostęp do zmiennej **l_kf** oraz, w celu symulacji pojedynczego kranu, do nalewania do kuflí.

```

for(i=0; i<ile_musze_wypic; i++){
    printf("\nKlient %d, wybieram kufel\n", moj_id);
    pthread_mutex_lock(&kf);
    l_kf = l_kf -1;
    pthread_mutex_unlock(&kf);

    pthread_mutex_lock(&kr);

```

```

printf("\nKlient %d, nalewam z kranu %d\n", moj_id, j);
usleep(30);
pthread_mutex_unlock(&kr);

printf("\nKlient %d, pije\n", moj_id);
nanosleep((struct timespec[]){{0, 50000000L}}, NULL);

pthread_mutex_lock(&kf);
printf("\nKlient %d, odkladam kufel\n", moj_id);
l_kf++;
pthread_mutex_unlock(&kf);

}

```

Kod ten nadal nie zabezpiecza przed sytuacją w której klientów jest więcej niż kufla, ale eliminuje on problem synchronizacji. Aby rozwiązać ten problem należy sprawdzić ilość dostępnych kufla po zamknięciu pierwszego mutexa, pamiętając o odblokowaniu go w wypadku braku kufla.

Zmodyfikowany fragment kodu z pobierania kufla z czekaniem na kufla:

```

while(1)
{
    pthread_mutex_lock(&kf);
    if(l_kf > 0)
    {
        l_kf = l_kf -1;
        pthread_mutex_unlock(&kf);
        break;
    }
    else
    {
        pthread_mutex_unlock(&kf);
        nanosleep((struct timespec[]){{0, 50000000L}}, NULL);
    }
}/**/

```

Rozwiązuje to problem nieistniejących kufla poprzez zwykły warunek **if()**. Pętla **while(1)** jest potrzebna aby klient próbował pobierać kufla do skutku.

Dalsze poprawianie pracy programu opiera się na zmianie funkcji **pthread_mutex_lock()** na **pthread_mutex_trylock()**, oraz na zliczaniu aktywnego oczekiwania klientów.

Dodanie *trylock* do pobierania kufla i zliczanie pracy (podobne modyfikacje dla oddawania kufla i rany, bez sprawdzania liczby kufla):

```

while(1)
{
    if(pthread_mutex_trylock(&kf) !=0)
    {
        if(l_kf > 0)
        {

```

```
printf("\nKlient %d, wybieram kufel\n", moj_id);
l_kf = l_kf -1;
pthread_mutex_unlock(&kf);
break;
}
else
{
    wykonana_praca++;
    pthread_mutex_unlock(&kf);
    nanosleep((struct timespec[]){{0, 50000000L}}, NULL);
}
}
else
{
    wykonana_praca++;
    praca++;
    nanosleep((struct timespec[]){{0, 50000000L}}, NULL);
}
}
```

Wynik ze zliczonym pracą:

```
Klient 3, wychodzę z pubu; wykonana praca 64
Liczba oczekiwanych 486
Liczba wolnych kuflów 10
Zamykamy pub!
```

laboratorium 5

Na tym laboratorium zadanie polegało na sprawdzenia czasu obliczania sumy tablicy w sposób sekwencyjny i równoległy, oraz na zbadaniu wpływu dokładności na obliczanie całki.

Przykładowe wywołanie **pthreads_suma**, czasy liczone w sekundach.

```
m@m-VirtualBox:~/PR_lab/lab_5/pthreads_suma$ ./pthreads_suma
Obliczenia sekwencyjne
0.000001
0.001606
0.000957
```

Wyniki pomiarów w sekundach:

[s]	równoległe		sekwencyjne	[s]	równoległe		sekwencyjne
wielkość tablicy	mutex	no mutex		wielkość tablicy	mutex	no mutex	
1000 000 opt	0,1123	0,102389	0,179003	1000 000 debug	0,209966	0,204781	0,396933
	0,107228	0,110678	0,178342		0,199631	0,226669	0,396398
	0,119204	0,0957	0,182518		0,223962	0,206683	0,380608
Średnia:	0,112911	0,102922	0,179954	Średnia:	0,211186	0,212718	0,391313

[s]	równoległe		sekwencyjne	[s]	równoległe		sekwencyjne
wielkość tablicy	mutex	no mutex		wielkość tablicy	mutex	no mutex	
1 000 opt	0,001532	0,000653	0,000002	1 000 debug	0,002054	0,000997	0,000003
	0,001973	0,001061	0,000002		0,003343	0,00631	0,000003
	0,001601	0,000755	0,000002		0,001744	0,000856	0,000003
Średnia:	0,001702	0,000823	0,000002	Średnia:	0,002380	0,002721	0,000003

Z pomiarów wynika że dla większych tablic zrównoleglenie w znaczący sposób przyśpieszyło program, a dla tablic mniejszych znaczco go spowolniło. Można za tym wnioskować tego że zrównoleglenie może być przydatnym narzędziem służącym do przyśpieszenia programu, jeżeli koszt związany ze zrównolegleniem (tworzenie i zarządzanie wątkami) jest mniejszy niż potencjalny zysk.

Pomiar dokładności, wartość teoretyczna to 2:

dx	0,1	0,01	0,0001
calka sin(x)	1,99839336097014	1,99839336097014	1,99999999833334
dx/wynik-2	0,01606639	0,160663903	1,66666E-05
dx	0,00001	0,0000001	
calka sin(x)	1,99999999998339	1,99999999999981	
dx/wynik-2	1,661E-06	1,9007E-06	

Zwiększenie dokładności zwiększa dokładność wyników, ale można zauważyc statę wydajności w zmniejszaniu dx(3 wiersz). Należy dobrać dokładność odpowiednią do konkretnego programu, tak aby nie zużywać niepotrzebnie zasobów, ale tak aby nadal otrzymać relatywnie poprawny wynik.