Sprawozdanie nr 2			
Przedmiot: Programowanie równoległe			
Imię i nazwisko: Monika Krakowska			
Temat: Wątki Pthreads			
Nr grupy: 4	Kierunek: Informatyka Techniczna	Nr laboratoriów: 3,4,5	Data oddania: 13.11.2024

Cele laboratoriów:

-Opanowanie tworzenia i implementacji programów równoległych z wykorzystaniem Pthreads.

-Nabycie praktycznych umiejętności zarządzania wątkami, w tym ich tworzenia, niszczenia i synchronizacji.

-Testowanie mechanizmów przekazywania argumentów do wątków.

-Poznanie sposobu działania obiektów definiujących atrybuty wątków.

Przebieg ćwiczeń:

1)Laboratorium 3:

-W pierwszym etapie ćwiczenia uzupełniłam według instrukcji kod programu pthreads_detach_kill.c pobranego ze strony przedmiotu.

-Kod działa następująco:

-wątek 1 zostaje utworzony, wątek główny daje mu 2s czasu na rozpoczęcie pracy, wątek potomny ustawia swój stan możliwości anulowania na wyłączony po czym następuje nieudana próba zabicia wątku. Kolejno po czasozajmowaczu następuje zmiana stanu możliwości anulowania na włączony i ponowna próba zabicia wątku. Sprawdzamy czy watek został zabity poprzez sprawdzenie czy wynik = pthread_canceled (wynik otrzymujemy z funkcji pthread_join).

//odpowiadający fragment kodu programu:

```
printf("watek glowny: tworzenie watku potomnego nr 1\n");
pthread_create(&tid, NULL, zadanie_watku, NULL); sleep(2);
// czas na uruchomienie watku
printf("\twatek glowny: wyslanie sygnalu zabicia watku\n");
pthread_cancel(tid);
pthread_join(tid, &wynik);
if (wynik == PTHREAD_CANCELED)
printf("\twatek glowny: watek potomny zostal zabity\n");
else printf("\twatek glowny: watek potomny NIE zostal zabity - blad\n");
```

//funkcja wątku

```
void*zadanie_watku(void*arg_wsk){
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
 printf("\twatek potomny: uniemozliwione zabicie\n");
 sleep(5);
 printf("\twatek potomny: umozliwienie zabicia\n");
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,NULL);
pthread_testcancel();
 zmienna_wspolna++;
 printf("\twatek potomny: zmiana wartosci zmiennej wspolnej\n");
 return(NULL); }
```

-wątek 2 tworzymy w taki sam sposób jak wątek 1, po czym następuje odłączenie go od wątku głównego poprzez funkcję pthread_detach, w konsekwencji czego wątek główny nie będzie na niego czekał. Anulujemy wątek funkcją pth_cancel. W tym momencie nie możemy bezpośrednio sprawdzić czy watek został anulowany tak jak w przypadku wątku 1, dlatego sprawdzamy wartość zmiennej wspólnej (pętla co 10 sekund sprawdza, czy wartość zmiennej się zmieniła – jeśli tak, oznacza to że wątek działa dalej, a jeśli nie – został prawdopodobnie zakończony)

//odpowiadający fragment kodu programu:

```
printf("watek glowny: tworzenie watku potomnego nr 2\n");
 pthread_create(&tid, NULL, zadanie_watku, NULL);
 sleep(2);
 // czas na uruchomienie watku printf("\twatek glowny: odlaczenie watku potomnego\n");
pthread_detach(tid);
 printf("\twatek glowny: wyslanie sygnalu zabicia watku odlaczonego\n");
 pthread_cancel(tid);
30 printf("\twatek glowny: czy watek potomny zostal zabity \n");
 printf("\twatek glowny: sprawdzanie wartosci zmiennej wspolnej\n"); for(i=0;i<10;i++);
sleep(10);
 if(zmienna_wspolna!=0) break;  }
 if (zmienna_wspolna==0)
 printf("\twatek glowny: odlaczony watek potomny PRAWDOPODOBNIE zostal zabity\n");
 else
 printf("\twatek glowny: odlaczony watek potomny PRAWDOPODOBNIE NIE zostal zabity\n");
```

-wątek 3 zostaje od razu utworzony jako odłączony (poprzez odpowiednie atrybuty). Wątek główny kończy swoją pracę, ale nie przerywa pracy wątku potomnego, który dalej działa w tle.

//odpowiadający fragment kodu programu:

```
pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 printf("watek glowny: tworzenie odlaczonego watku potomnego nr 3\n");
pthread_create(&tid, &attr, zadanie_watku, NULL);
 pthread_attr_destroy(&attr);
printf("\twatek glowny: koniec pracy, watek odlaczony pracuje dalej\n");
pthread_exit(NULL); }
```

-W drugim etapie ćwiczeń napisałam nową procedurę wątków, do której jako argument przesyłany jest identyfikator każdego wątku i która wypisuje systemowy ID wątku oraz swój przesłany identyfikator.

//odpowiadający fragment kodu programu:

```
void *funkcja_watku(void *arg) {
 int ID_watku = *(int *)arg;
 printf("Watek o ID: %d, systemowy ID: %lu\n", ID_watku, pthread_self());
 pthread_exit(NULL);
}
int main() {
 int liczba_watkow;
printf("Podaj liczbe watkow: ")
; scanf("%d", &liczba_watkow);
 pthread_t watki[liczba_watkow];
int watki_id[liczba_watkow];
for (int i = 0; i < liczba_watkow; i++) {
pthread_create(&watki[i], NULL, funkcja_watku, (void*)&watki_id[i]);
}
for (int i = 0; i < liczba_watkow; i++) {
pthread_join(watki[i], NULL);
}
 printf("\nWszystkie watki zakonczyly dzialanie.\n\n");
```

```
return 0;
}
```

Ważne jest, aby w celu poprawnego przesyłania identyfikatorów do wątków, stosować indywidualne zmienne dla każdego wątku lub dynamicznie alokować pamięć, aby zapewnić izolację danych i uniknąć błędów synchronizacji. W załącznikach znajdują się wydruki z terminala z wersją poprawnie działającą oraz taką, gdzie pojawił się błąd synchronizacji (gdzie zamiast unikalnych id- „watki_id[]", została przesłana zmienna „i").

-W ostatnim etapie mojego ćwiczenia napisałam nowy program, w którym każdy wątek jako argument dostaje wskaźnik do swojej struktury z danymi wejściowymi, program dzieli przedział na fragmenty i przydziela je do wątków, struktura zawiera współrzędne początku i końca fragmentu oraz miejsce na wynik, każdy wątek oblicza średnią, a główna funkcja sumuje wyniki i wypisuje sumę.

//odpowiadający fragment kodu programu:

//struktura:

```
typedef struct {
  double start; // Początek fragmentu przedziału (in)
   double end;   // Koniec fragmentu przedziału (in)
 double result; // Wynik (in/out)
} WatekData;
```

//funkcja wątku:

```
void *funkcja_watku(void *arg) {
   WatekData *data = (WatekData *)arg;
   data->result = (data->start + data->end) / 2.0;
   printf("Watek obliczyl srednia dla przedzialu (%.2f, %.2f): %.2f\n", data->start, data->end, data->result);
   pthread_exit(NULL);
}
```

//fragment funkcji main():

```
double dlugosc_fragmentu = (b - a) / liczba_watkow;
   for (int i = 0; i < liczba_watkow; i++) {
      dane[i].start = a + i * dlugosc_fragmentu;
      dane[i].end = dane[i].start + dlugosc_fragmentu;
      dane[i].result = 0.0; // Inicjalizacja wyniku
   pthread_create(&watki[i], NULL, funkcja_watku, (void *)&dane[i]);
```

```
}
for (int i = 0; i < liczba_watkow; i++) {
  pthread_join(watki[i], NULL);
}
double suma_wynikow = 0.0;
for (int i = 0; i < liczba_watkow; i++) {
  suma_wynikow += dane[i].result;
}
printf("\nSuma wynikow: %.2f\n", suma_wynikow);
```

1)Dodatkowe wnioski i odpowiedzi na pytania:

Wątki pthreads mogą działać w dwóch trybach: połączonym (joinable), czyli standardowym, gdzie wątek główny może czekać na zakończenie działania wątku potomnego, można zbierać informację o jego zakończeniu (funkcja pthread_join) lub w trybie odłączonym (detached), gdzie wątki działają niezależnie, nie można na nie czekać i nie ma bezpośredniego mechanizmu na sprawdzenie czy wątek został zakończony.

Standardowo wątek kończy swoje działanie w momencie, gdy funkcja, która wykonuje dobiegnie końca i zwróci wynik (lub wywoła pthread_exit()). Alternatywnie wątek może zostać anulowany poprzez pthread_cancel().

Watek może się „chronić” przed próbą zabicia poprzez ustawienie swojego stanu anulowania na wyłączony pthread_setcancelstate (PTH_CANCEL_DISABLE,NULL). W przypadku wątków połączonych można łatwo sprawdzić czy wątek został faktycznie zabity, czekając na niego i zbierając wynik (pthread_join(tid,&wynik), a następnie sprawdzając (if (wynik = pthread_canceled). W przypadku wątków odłączonych nie mamy tak prostego mechanizmu sprawdzania czy wątek został anulowany, dlatego w naszym przypadku sprawdzamy czy wątek dalej działa i na tej podstawie określamy czy wątek żyje czy prawdopodobnie zakończył swoje działanie.

2)Laboratorium 4:

-W trakcie laboratorium nr 2 przeprowadziliśmy symulację pubu z różnymi mechanizmami „zabierania” kufla:

-w pierwszej wersji kufle reprezentowała zwykła zmienna globalna równa ilości dostępnych kufli w barze. Klient zabierając kufel zmniejszał ilość dostępnych kufli poprzez: dostępne_kufle--, a następnie odkładając zwiększał ją: dostępne_kufle++. Wersja ta doprowadzała do wielu błędów, w konsekwencji których klienci mimo braku wolnych kufli wciąż „zabierali” je, a ostateczna ilość kufli nie była zgodna z tą początkową.

-w drugiej wersji programu dodaliśmy procedury pthread_mutex_lock oraz pthread_mutex_unlock wykorzystując wciąż naszą zmienna globalną dostępne_kufle. Zamykamy mutexa przed zmienieniem wartości zmiennej dostępne_kufle i otwieramy go zaraz po. Ta wersja również doprowadzała do tego samego typu błędów co poprzednia.

-w trzeciej wersji programu zaimplementowałam wariant aktywnego czekania na kufle tzw.busy waiting. W tym podejściu klient sprawdza dostępność kufla i dopiero go pobiera lub czeka określoną ilość sekund, jeśli żaden kufel nie jest dostępny. Dodatkowo użyłam zmiennej success pozwalającej na zakończenie pętli sprawdzającej dostępność kufla, gdy klient go zdobędzie. Ten program zadziałał poprawnie.

//odpowiedni fragment kodu:

```
do {
 pthread_mutex_lock(&mutex);
if (dostepne_kufle > 0)
{ dostepne_kufle--;
 success = 1;
printf("Klient %d pobrał kufel. Dostępne kufle: %d\n", klient_id, dostepne_kufle);
 } pthread_mutex_unlock(&mutex);
 if (!success) {
printf("Klient %d czeka na dostępny kufel.\n", klient_id);
 sleep(10);
 CPU }
 } while (!success);
```

-w ostatniej wersji programu dodałam funkcję pthread_mutex_trylock, która pozwoliła uniknąć oczekiwania w przypadku zajętego mutexu. Zamiast blokować wątek, trylock zwraca natychmiast, dzięki czemu wątek może przejść do innego zadania i ponowić próbę dostępu do sekcji krytycznej później. Dodatkowo wprowadziłam zmienną zliczającą „pracę" wątku czyli ilość wykonywania, tego „innego zadania" (u nas było to po prostu sleep(8)) .

2) Dodatkowe wnioski i odpowiedzi na pytania:

Rozwiązanie problemu bezpiecznego korzystania z kufli wymaga zastosowania pthread_mutex_lock w celu ochrony sekcji krytycznej, w której sprawdzana i aktualizowana będzie liczba dostępnych kufli; sprawdzenia dostępności kufli w tejże sekcji oraz zwolnienia mutexu po zakończonej operacji, aby umożliwić innym wątkom dostęp do sekcji.

Zastosowanie pthread_mutex_trylock pozwala uniknąć blokowania wątku, co sprawdza się, jeśli chcemy, aby wątki mogły wykonywać inne zadania podczas oczekiwania. Jeśli chodzi o ilość pracy to zależy ona od liczby klientów i proporcji liczby kufli do liczby klientów. Wadą trylock jest jednak to, że aktywne czekanie nadal zużywa zasoby procesora. Lepszym podejściem byłoby użycie bardziej zaawansowanych mechanizmów synchronizacji, które pozwalają na usypianie wątku do momentu, aż zasób stanie się dostępny.

3) Laboratorium 5:

- w trakcie laboratorium 5 przeprowadziłam analizę dwóch podejść do zrównoleglenia pobranego programu suma (tablica, i zmienna globalna) oraz przeprowadziłam pomiary czasu dla wersji sekwencyjnej i równoległej. Wątek dostaje przedział z którego ma obliczyć sumę. W podejściu ze zmienną globalną, każdy wątek czeka na swoją kolej aby do zmiennej globalnej z wynikiem wszystkich sum dopisać swój wynik. W podejściu z tablica każdy wątek zapisuje wynik w innym miejscu tablicy, a po ich zakończeniu wszystkie elementy tablicy są sumowane, dzięki czemu wszystkie wątki działają równolegle i żaden nie musi czekać na „swoją kolej".

Otrzymane wyniki pomiarów:

	tab 100 000 000 -03				tab 100 000 000 -g	
l. wątków	czas	wariant		l. wątków	czas	wariant
1	0,189798	-		1	0,498473	-
1	0,189672	-		1	0,628098	-
1	0,193623	-		1	0,614897	-
2	0,08872	mutex		2	0,303297	mutex
2	0,097107	mutex		2	0,300782	mutex
2	0,089862	mutex		2	0,316276	mutex
2	0,07345	tablica		2	0,236067	tablica
2	0,077739	tablica		2	0,250574	tablica
2	0,073036	tablica		2	0,271907	tablica

	tab 1000 -03				tab 1000 -g	
l. wątków	czas	wariant		l. wątków	czas	wariant
1	0,000001	-		1	0,000005	-
1	0,000001	-		1	0,000005	-
1	0,000001	-		1	0,000004	-
2	0,001641	mutex		2	0,001551	mutex
2	0,00214	mutex		2	0,002533	mutex
2	0,002028	mutex		2	0,002578	mutex

2	0,000273	tablica		2	0,000369	tablica
2	0,000201	tablica		2	0,000353	tablica
2	0,000116	tablica		2	0,000375	tablica

Zgodnie z moimi oczekiwaniami szybsze okazało się podejście z tablicami, oraz oczywiście wersja z opcją optymalizacji od tej z opcją debuggowania. Przy małych tablicach wersja bez synchronizacji okazała się szybsza, jest natomiast bardzo ryzykowna. Różnice nie są aż tak duże, jak się tego spodziewałam i widać je bardziej na małych tablicach, co mnie zdziwiło.

3) Dodatkowe wnioski i odpowiedzi na pytania:

Zrównoleglenie pętli może prowadzić do przyspieszenia obliczeń w przypadku dużych zadań, ale ma także swoje wady, takie jak narzut związany z synchronizacją (mutex) oraz dodatkowe zużycie pamięci. W przypadku małych zadań, narzut związany z tworzeniem wątków oraz synchronizacją może powodować, że zrównoleglenie nie przynosi korzyści, a może wręcz spowolnić obliczenia.

Załącznik 1. Wydruk z terminala – działanie programu pthreads_detach_kill:

```
ubuntu@ubuntu:~/Desktop/PR_lab/lab_3/zad_1$ ./pthreads_detach_kill
watek glowny: tworzenie watku potomnego nr 1
        watek potomny: uniemozliwione zabicie
        watek glowny: wyslanie sygnalu zabicia watku
        watek potomny: umozliwienie zabicia
        watek glowny: watek potomny zostal zabity
watek glowny: tworzenie watku potomnego nr 2
        watek potomny: uniemozliwione zabicie
        watek glowny: odlaczenie watku potomnego
        watek glowny: wyslanie sygnalu zabicia watku odlaczonego
        watek glowny: czy watek zostal zabity
        watek glowny: sprawdzenie wartosci zmiennej wspolnej
        watek potomny: umozliwienie zabicia
        watek glowny: odlaczony watek potomny prawdopodobnie zostal zabity
watek glowny: tworzenie odlaczonego watku potomnego nr 3
        watek glowny: koniec pracy, watek odlaczony pracuje dalej
        watek potomny: uniemozliwione zabicie
        watek potomny: umozliwienie zabicia
        watek potomny: zmiana wartosci zmiennej wspolnej
```

Załącznik 3. Wydruk z terminala – niepoprawne działanie programu watki (powtarzające się ID):

```
ubuntu@ubuntu:~/Desktop/lab$ gcc watki.c -o watki
ubuntu@ubuntu:~/Desktop/lab$ ./watki
Podaj liczbe watkow: 6
Watek o ID: 1, systemowy ID: 133043473876672
Watek o ID: 2, systemowy ID: 133043463390912
Watek o ID: 6, systemowy ID: 133043452905152
Watek o ID: 6, systemowy ID: 133043442419392
Watek o ID: 6, systemowy ID: 133043431933632
Watek o ID: 6, systemowy ID: 133043421447872

Wszystkie watki zakonczyly dzialanie.
```

Załącznik 3. Wydruk z terminala – poprawne działanie programu watki (unikalne ID):

```
ubuntu@ubuntu:~/Desktop/lab$ gcc watki.c -o watki
ubuntu@ubuntu:~/Desktop/lab$ ./watki
Podaj liczbe watkow: 6
Watek o ID: 0, systemowy ID: 128157967451840
Watek o ID: 1, systemowy ID: 128157956966080
Watek o ID: 2, systemowy ID: 128157946480320
Watek o ID: 3, systemowy ID: 128157935994560
Watek o ID: 4, systemowy ID: 128157925508800
Watek o ID: 5, systemowy ID: 128157915023040

Wszystkie watki zakonczyly dzialanie.

ubuntu@ubuntu:~/Desktop/lab$
```