Programowanie równoległe	Kierunek: Informatyka Techniczna	Grupa: 9
Imię i nazwisko: Karolina Starzec	Temat: LAB 2 i 3	Termin oddania: 27.10.2024

LAB 2

1. Cel

Przeprowadzenie pomiaru czasu CPU i zegarowego tworzenia procesów i wątków systemowych Linuxa oraz nabycie umiejętności pisania programów wykorzystujących tworzenie wątków i procesów.

2. Realizacja zajęć

Realizując zajęcia skopiowałam oraz rozpakowałam potrzebne pliki ze strony przedmiotu oraz przeczytałam uwagi na temat pomiaru czasu na stronie przedmiotu. Uzupełniłam pliki fork.c oraz clone.c o procedury pomiaru czasu inicjuj_czas() i drukuj_czas(). Przeprowadziłam test dla wersji bez optymalizacji (-g -DDEBUG -O0) i z optymalizacją (-O3).

fork.c:

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

#include"../pomiar_czasu/pomiar_czasu.h"

int zmienna_globalna=0;

int main(){

  int pid, wynik, i;

  inicjuj_czas();

  for(i=0;i<10000;i++){

    pid = fork();

    if(pid==0){

      zmienna_globalna++;

      exit(0);

    } else {

      wait(NULL);
    }
  }
  drukuj_czas();
}
```

Wyniki bez optymalizacji:

	Próba 1	Próba 2	Próba 3	Średnia
Czas standardowy	1.230145	1.284978	1.290675	1.268599
Czas CPU	0.007351	0.018774	0.014536	0.013355
Czas zegarowy	3.135035	3.298804	3.317897	3.249479

Wyniki z optymalizacją:

	Próba 1	Próba 2	Próba 3	Średnia
Czas standardowy	1.313235	1.257393	1.281823	1.284150
Czas CPU	0.011773	0.019004	0.024763	0.018513
Czas zegarowy	3.329342	3.261716	3.279473	3.290177

clone.c

```c
#define _GNU_SOURCE
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sched.h>
#include <linux/sched.h>
#include "../pomiar_czasu/pomiar_czasu.h"

int zmienna_globalna=0;
#define ROZMIAR_STOSU 1024*64

int funkcja_watku( void* argument )
{
  zmienna_globalna++;

  return 0;
}

int main()
{
  void *stos;
  pid_t pid;
  int i;

  stos = malloc( ROZMIAR_STOSU );
  if (stos == 0) {
    printf("Proces nadrzędny - blad alokacji stosu\n");
    exit( 1 );
  }
```

```c
  for(i=0;i<10000;i++){

    pid = clone( &funkcja_watku, (void *) stos+ROZMIAR_STOSU,
                 CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, 0 );

    waitpid(pid, NULL, __WCLONE);
  }

drukuj_czas(); //to tez

  free( stos );
}
```

Wyniki bez optymalizacji:

	Próba 1	Próba 2	Próba 3	Średnia
Czas standardowy	0.318252	0.312396	0.309662	3.123437
Czas CPU	0.010616	0.000000	0.004394	0.005003
Czas zegarowy	0.631363	0.624365	0.612435	0.622721

Wyniki z optymalizacją:

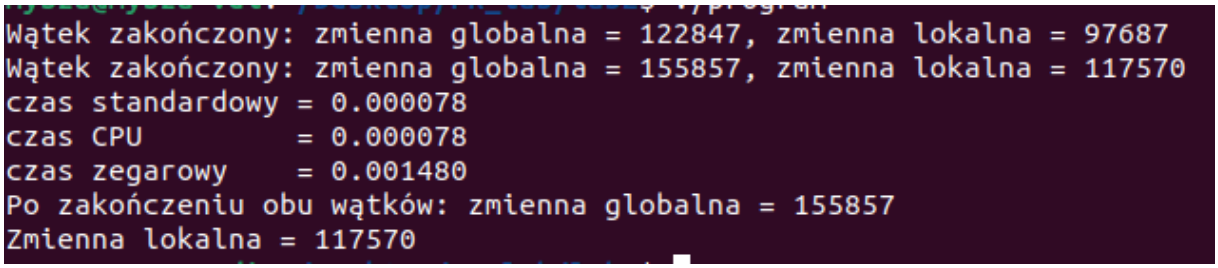
	Próba 1	Próba 2	Próba 3	Średnia
Czas standardowy	0.311926	0.311316	0.311229	0.311490
Czas CPU	0.010316	0.003663	0.015742	0.009907
Czas zegarowy	0.626031	0.620566	0.628309	0.624968

Następnie napisałam nowy program (program.c) na podstawie clone.c, tym razem bez tworzenia pętli wątków, w którym bezpośrednio po sobie tworzone są dwa wątkido działania równoległego.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sched.h>
#include <linux/sched.h>
#include "../pomiar_czasu/pomiar_czasu.h"

#define ROZMIAR_STOSU 1024*64
#define LICZBA_ITERACJI 100000


int zmienna_globalna = 0;


int funkcja_watku(void *arg)
{
    int *zmienna_lokalna = (int *)arg;

    for (int i = 0; i < LICZBA_ITERACJI; i++) {
        zmienna_globalna++;
        (*zmienna_lokalna)++;
    }

    printf("Wątek zakończony: zmienna globalna = %d, zmienna lokalna = %d\n", zmienna_globalna, *zmienna_lokalna);

    return 0;
}

int main()
{
    void *stos1, *stos2;
    pid_t pid1, pid2;

    int zmienna_lokalna = 0;
```

```c
    stos1 = malloc(ROZMIAR_STOSU);
    stos2 = malloc(ROZMIAR_STOSU);
    if (stos1 == NULL || stos2 == NULL) {
        printf("Błąd alokacji stosu\n");
        exit(1);
    }


    inicjuj_czas();

    pid1 = clone(funkcja_watku, (void *)stos1 + ROZMIAR_STOSU,
                 CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, &zmienna_lokalna);
    if (pid1 == -1) {
        printf("Błąd tworzenia pierwszego wątku\n");
        exit(1);
    }

    pid2 = clone(funkcja_watku, (void *)stos2 + ROZMIAR_STOSU,
                 CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, &zmienna_lokalna);
    if (pid2 == -1) {
        printf("Błąd tworzenia drugiego wątku\n");
        exit(1);
    }

    waitpid(pid1, NULL, __WCLONE);
    waitpid(pid2, NULL, __WCLONE);

    free(stos1);
    free(stos2);

    drukuj_czas();

    printf("Po zakończeniu obu wątków: zmienna globalna = %d\n", zmienna_globalna);
    printf("Zmienna lokalna = %d\n", zmienna_lokalna);

    return 0;
}
```

Wyniki:



3. Wnioski do plików fork.c i clone.c

Relatywna wielkość narzutu na tworzenie wątków w porównaniu do tworzenia procesów:

W przypadku fork: średni czas zegarowy dla operacji bez optymalizacji to około 3.25 sekundy, natomiast z optymalizacją również jest zbliżony do tej wartości (około 3.29 sekundy).

W przypadku clone: średni czas zegarowy bez optymalizacji wynosi około 0.62 sekundy, a z optymalizacją również jest zbliżony do tej wartości (około 0.625 sekundy).

Narzut na tworzenie procesu za pomocą „fork” jest znacznie większy niż na tworzenie wątku za pomocą „clone”. Zależność ta jest spowodowana różnicą w operacjach, ponieważ „fork” tworzy osobny proces, co wymaga skopiowania zasobów procesu rodzica, takich jak przestrzeń pamięci, podczas gdy „clone” współdzieli pamięć między wątkami, co jest mniej zasobożerne.

Szacowana liczba operacji arytmetycznych, które może wykonać procesor w czasie tworzenia wątku

Zakładając, że procesor działa z częstotliwością np. 2 GHz, możemy oszacować liczbę operacji, które procesor może wykonać:

*Liczba operacji = czas tworzenia wątku * częstotliwość procesora*

Średni czas tworzenia wątku wynosi około 0.62 sekundy, więc liczba operacji wynosi:

0.62 (s) * 2 * 10^9 (Hz) = 1.24 * 10^9 operacji

Procesor mógłby wykonać około 1.24 miliarda operacji arytmetycznych w czasie potrzebnym do utworzenia wątku.

Liczba operacji wejścia/wyjścia

Liczba operacji I/O zależy od typu operacji i przepustowości magistrali, ale można założyć, że procesor w czasie tworzenia wątku może wykonać kilkaset operacji I/O (dla powolnych urządzeń, jak dyski, mniej; dla operacji w pamięci może to być o wiele więcej).

Wpływ optymalizacji na czas tworzenia procesów i wątków

Z wyników wynika, że optymalizacja `-O3` nie wpływa znacząco na czas tworzenia zarówno procesów, jak i wątków. Zarówno dla „fork”, jak i dla „clone”, czasy zegarowe z optymalizacją i bez niej są bardzo zbliżone. Przyczyną tego jest fakt, że operacje „fork” i „clone” to wywołania systemowe, a nie zależą od kodu użytkownika, więc optymalizacja kodu nie ma tutaj wpływu na ich wydajność.

4. Wnioski do pliku program.c

Czy wartości zmiennych odpowiadają liczbie operacji wykonanych przez wątki?

Zmienna globalna: Ponieważ każdy z wątków dodaje 1 do tej samej zmiennej globalnej w każdej iteracji, po wykonaniu 100,000 iteracji przez dwa wątki powinna mieć wartość 100,000 × 2 = 200,000, jednak wynosi 155,857.

Zmienna lokalna: Każdy wątek powinien modyfikować swoją wersję zmiennej lokalnej przekazanej przez wskaźnik. W idealnych warunkach, gdyby zmienna lokalna była osobna dla każdego wątku, powinna wynosić 100,000 dla każdego wątku. Wartości zmiennej lokalnej w obu wątkach nie wynoszą dokładnie 100,000, lecz 97687 i 117,570.

Warunki wyścigu na zmiennych globalnych i lokalnych są odpowiedzialne za niespójne wyniki. Bez synchronizacji oba wątki odczytują i modyfikują zmienne równocześnie, prowadząc do niespójności.

Zachowanie zmiennych globalnych i lokalnych w wątkach

Zmienne globalne: Są współdzielone między wątkami. Każdy z wątków bez synchronizacji odczytuje, modyfikuje i zapisuje tę samą wartość zmienna_globalna. W efekcie dochodzi do warunków wyścigu, czyli operacje dodawania są przeplatane i nadpisywane, co skutkuje nieoczekiwanymi wartościami.

Zmienne lokalne: Teoretycznie zmienna lokalna jest przekazywana jako argument do funkcji każdego wątku. W praktyce, ponieważ oba wątki operują na tej samej zmiennej, również występuje tutaj wyścig danych, co prowadzi do niespójnych wartości na końcu pracy wątków. Każdy wątek odczytuje, modyfikuje i zapisuje zmienna_lokalna bez świadomości zmian wprowadzonych przez inny wątek.

LAB 3

1. Cel

Celem ćwiczenia jest nabycie praktycznej umiejętności manipulowania wątkami Pthreads, a dokładnie tworzenia, niszczenia i elementarnej synchronizacji, przetestowanie mechanizmu przesyłania argumentów do wątku oraz poznanie funkcjonowania obiektów określających atrybuty wątków.

2. Realizacja zajęć

Realizując tą część zajęć skopiowałam oraz rozpakowałam potrzebne pliki ze strony przedmiotu. Następnie uzupełniłam kod programu pthreads_detach_kill.c we wskazanych miejscach.

```c
int main() {
    pthread_t tid;
    pthread_attr_t attr;
    void *wynik;
    int i;

    // Watek przyłączalny
    printf("watek glowny: tworzenie watku potomnego nr 1\n");

    // Tworzenie wątku z domyślnymi właściwościami
    pthread_create(&tid, NULL, zadanie_watku, NULL);

    sleep(2); // czas na uruchomienie wątku

    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");

    // Odłączanie wątku
    zmienna_wspolna = 0;

    printf("watek glowny: tworzenie watku potomnego nr 2\n");

    // Tworzenie wątku z domyślnymi właściwościami
    pthread_create(&tid, NULL, zadanie_watku, NULL);

    sleep(2); // czas na uruchomienie wątku

    printf("\twatek glowny: odlaczenie watku potomnego\n");
    pthread_detach(tid); // Instrukcja odłączenia

    printf("\twatek glowny: wyslanie sygnalu zabicia watku odlaczonego\n");
    pthread_cancel(tid);

    printf("\twatek glowny: czy watek potomny zostal zabity \n");
    printf("\twatek glowny: sprawdzanie wartosci zmiennej wspolnej\n");
    for (i = 0; i < 10; i++) {
        sleep(1);
        if (zmienna_wspolna != 0) break;
    }

    if (zmienna_wspolna == 0)
        printf("\twatek glowny: odlaczony watek potomny PRAWODOPOBNIE zostal zabity\n");
    else
        printf("\twatek glowny: odlaczony watek potomny PRAWODOPOBNIE NIE zostal zabity\n");

    // Watek odłączony
    pthread_attr_init(&attr); // Inicjacja atrybutów
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); // Ustawianie typu wątku na odłączony

    printf("watek glowny: tworzenie odlaczonego watku potomnego nr 3\n");
    pthread_create(&tid, &attr, zadanie_watku, NULL);

    pthread_attr_destroy(&attr); // Niszczenie atrybutów

    printf("\twatek glowny: koniec pracy, watek odlaczony pracuje dalej\n");
    pthread_exit(NULL); // co stanie sie gdy uzyjemy exit(0)?
}
```

Wynik programu:

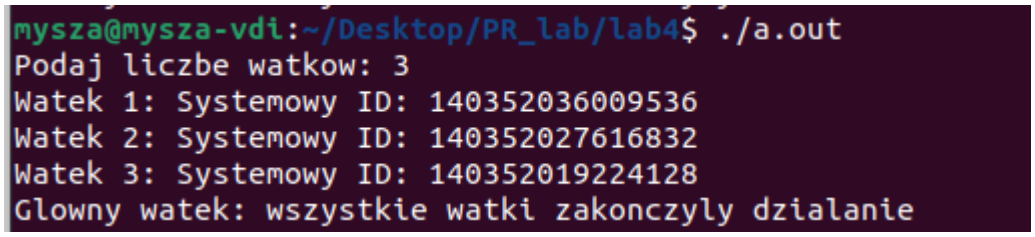
```
mysza@mysza-vdi:~/Desktop/PR_lab/lab4/zad1$ ./a.out
watek glowny: tworzenie watku potomnego nr 1
        watek potomny: uniemozliwione zabicie
        watek glowny: wyslanie sygnalu zabicia watku
        watek potomny: umozliwienie zabicia
        watek glowny: watek potomny zostal zabity
watek glowny: 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 potomny zostal zabity
        watek glowny: sprawdzanie wartosci zmiennej wspolnej
        watek potomny: umozliwienie zabicia
        watek glowny: odlaczony watek potomny PRAWODOPOBNIE 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
```

Następnie samodzielnie utworzyłam nową procedurę wątków, do której jako argument przesyłany jest identyfikator każdego wątku, z zakresu 0..liczba_wątków-1.

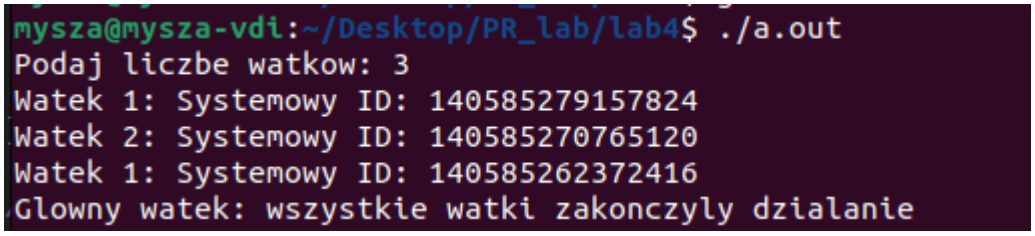
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

// Funkcja wykonywana przez każdy wątek
void* zadanie_watku(void* arg) {
    int id = *((int*)arg); // Pobierz identyfikator wątku
    printf("Watek %d: Systemowy ID: %lu\n", id, pthread_self());
    return NULL;
}

int main() {
    int liczba_watkow;

    // Pytanie użytkownika o liczbę wątków
    printf("Podaj liczbe watkow: ");
    scanf("%d", &liczba_watkow);

    // Tworzymy tablicę wątków
    pthread_t watki[liczba_watkow];
    int id[liczba_watkow];

    // Tworzenie wątków
    for (int i = 0; i < liczba_watkow; i++) {
        id[i] = i; // Każdy wątek otrzymuje swój identyfikator
        if (pthread_create(&watki[i], NULL, zadanie_watku, &id[i]) != 0)
            perror("Blad przy tworzeniu watku");
            exit(EXIT_FAILURE);
        }
    }
```

```c

    // Oczekiwanie na zakończenie każdego z wątków
    for (int i = 0; i < liczba_watkow; i++) {
        if (pthread_join(watki[i], NULL) != 0) {
            perror("Blad przy oczekiwaniu na zakonczenie watku");
            exit(EXIT_FAILURE);
        }
    }

    printf("Glowny watek: wszystkie watki zakonczyly dzialanie\n");
    return 0;
}
```

Wynik bez błędu:



Wynik z błędem:



3. Wnioski do programu pthreads_detach_kill.c

W jakich dwóch trybach mogą funkcjonować wątki Pthreads? Jaka jest różnica między tymi trybami? W jaki sposób można wymusić zakończenie działania wątku?

Tryb przyłączony: Pozwala dołączyć wątek do głównego wątku za pomocą pthread_join, co umożliwia synchronizację jego zakończenia i uzyskanie zwracanej wartości. Wątek kończy działanie po wykonaniu funkcji lub po anulowaniu przez pthread_cancel, a pthread_join zwraca PTHREAD_CANCELED, jeśli anulacja się powiodła.

Tryb odłączony: Wątki odłączone nie mogą być dołączone, a ich zakończeniem zarządza system. Nie wymagają pthread_join, ale nie pozwalają sprawdzić, czy zakończyły się poprawnie lub zostały anulowane.

Jak wątek może chronić się przed próbą "zabicia"?

Wątek może zabezpieczyć się przed próbą zabicia przez ustawienie swojego stanu anulowania za pomocą funkcji pthread_setcancelstate.

Jak można sprawdzić czy próba "zabicia" wątku powiodła się?

Dla wątków przyłączonych można sprawdzić, czy anulowanie się powiodło, poprzez sprawdzenie wartości zwróconej przez pthread_join. Jeśli wątek został anulowany, pthread_join zwróci PTHREAD_CANCELED. W przypadku wątków odłączonych taka informacja nie jest dostępna, ponieważ system automatycznie zarządza ich zakończeniem i zwalnianiem zasobów, więc nie ma możliwości sprawdzenia, czy anulowanie się powiodło.

4. Wnioski do programu zad.c

W jaki sposób można poprawnie przesłać identyfikator do wątku? Jaki może pojawić się błąd synchronizacji w przypadku próby przesłania zwykłego wskaźnika do liczby całkowitej?

Aby poprawnie przesłać identyfikator do każdego wątku w kodzie, należy użyć oddzielnej zmiennej dla każdego wątku. Najlepiej użyć tablicy do przechowywania identyfikatorów wątków. Jeśli jednak przekażemy do funkcji wątków wskaźnik do jednej zmiennej sterującej pętli, pojawi się błąd synchronizacji. Wynika to z faktu, że zanim wątek odczyta tę wartość, pętla może już zmienić zmienną sterującą na wartość przypisaną kolejnemu wątkowi.