Programowanie Równoległe	Kierunek: Informatyka Techniczna	Grupa: 9
Imię i nazwisko: Jakub Świerczyński	Temat: Lab 2, lab 3	Termin oddania: 27.10.2024

Lab 2

Cel laboratorium

Celem laboratorium było przeprowadzenie pomiaru czasu CPU oraz czasu zegarowego tworzenia procesów i wątków systemowych w środowisku Linux. Zadanie obejmowało również rozwinięcie umiejętności programowania z użyciem procesów i wątków oraz analizę relatywnych kosztów czasowych operacji związanych z ich tworzeniem. Dodatkowym celem było porównanie narzutu czasowego wersji zoptymalizowanej i nieoptymalizowanej kodu.

Realizacja zadania

Na początku utworzono katalog roboczy lab_2 oraz pobrano i rozpakowano pliki fork_clone.tgz. Programy przygotowano z użyciem kompilatora gcc i standardowego pliku Makefile z dwiema opcjami kompilacji:

- Wersja debugowania: bez optymalizacji (-g -DDEBUG -O0)

- Wersja zoptymalizowana: z optymalizacją czasową (-O3)

Fork

Bez optymalizacji

Wersja	Czas standardowy	Czas CPU	Czas zegarowy
Test 1	0,547235	0,039941	1,345057
Test 2	0,532043	0,018448	1,336050
Test 3	0,553569	0,029614	1,356906
Test 4	0,550389	0,019963	1,328634
Średnie	0,545809	0,026992	1,341662

Z optymalizacją

Wersja	Czas standardowy	Czas CPU	Czas zegarowy
Test 1	0,533553	0,018471	1,283684
Test 2	0,550934	0,031226	1,339485
Test 3	0,539121	0,017171	1,332914
Test 4	0,533457	0,010900	1,312487
Średnie	0,53926625	0,019442	1,317143

Clone

Bez optymalizacji

Wersja	Czas standardowy	Czas CPU	Czas zegarowy
Test 1	0,278506	0,011542	0,519876
Test 2	0,298269	0,018705	0,563114
Test 3	0,285475	0,009959	0,523116
Test 4	0,261793	0,028576	0,492504
Średnie	0,28101075	0,0171955	0,5246525

Z optymalizacją

Wersja	Czas standardowy	Czas CPU	Czas zegarowy
Test 1	0,272396	0,010369	0,513483
Test 2	0,291543	0,008716	0,562252
Test 3	0,292739	0,009819	0,57787
Test 4	0,265949	0,010255	0,487734
Średnie	0,28065675	0,00978975	0,53533475

Program

Bez optymalizacji

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab2# ./program
Wątek zakończył. Zmienna globalna: 127707, Zmienna lokalna: 127707
Wątek zakończył. Zmienna globalna: 127707, Zmienna lokalna: 127707
136894, Zmienna lokalna: 133375
Main zakończył. Zmienna globalna: 136894
Main zakończył. Zmienna lokalna 1: 133375
Main zakończył. Zmienna lokalna 2: 133375
czas standardowy = 0.000107
czas CPU         = 0.000107
czas zegarowy    = 0.003743
```

Wersja	Czas standardowy	Czas CPU	Czas zegarowy
Test 1	0,000092	0,000091	0,001581
Test 2	0,000142	0,000155	0,003416
Test 3	0,000097	0,000096	0,000613
Test 4	0,000107	0,000107	0,003743
Średnie	0,0001095	0,0001123	0,00233825

Z optymalizacją

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab2# ./program
Wątek zakończył. Zmienna globalna: 100218, Zmienna lokalna: 100264
Wątek zakończył. Zmienna globalna: 192023, Zmienna lokalna: 192069
Main zakończył. Zmienna globalna: 192023
Main zakończył. Zmienna lokalna 1: 192069
Main zakończył. Zmienna lokalna 2: 192069
czas standardowy = 0.000165
czas CPU         = 0.000165
czas zegarowy    = 0.000967
```

Wersja	Czas standardowy	Czas CPU	Czas zegarowy
Test 1	0,000108	0,000108	0,001079
Test 2	0,000118	0,000119	0,003228
Test 3	0,000106	0,000107	0,004248
Test 4	0,000165	0,000165	0,000967
Średnie	0,00012425	0,00012475	0,0023805

Analiza oraz wnioski dla fork i clone

Analiza wyników dla fork (tworzenie procesów)

Bez optymalizacji:

- Średni czas zegarowy wynosi 1,3417 s, a średni czas CPU 0,0270 s.

- Wersja bez optymalizacji jest bardziej czasochłonna zarówno pod względem zegarowym, jak i CPU.

Z optymalizacją:

- Średni czas zegarowy wynosi 1,3171 s, co wskazuje na nieznaczne zmniejszenie czasu zegarowego o około 0,0246 s w stosunku do wersji bez optymalizacji.

- Średni czas CPU zmniejszył się do 0,0194 s, co stanowi redukcję o około 28% w porównaniu z wersją bez optymalizacji.

Wnioski dla fork

- Niewielkie przyspieszenie zegarowe: Optymalizacja zmniejszyła czas zegarowy o 1,8%, co oznacza, że wpływ optymalizacji na całkowity czas operacji tworzenia procesów jest ograniczony.

- Zauważalna oszczędność CPU: Optymalizacja znacząco zmniejszyła czas CPU o 28%, co wskazuje, że dla procesu fork kompilator potrafi zoptymalizować operacje tak, aby zużywały mniej zasobów CPU, nawet jeśli wpływ na czas zegarowy jest ograniczony.

Analiza wyników dla clone (tworzenie wątków)

Bez optymalizacji:

- Średni czas zegarowy wynosi 0,5247 s, a średni czas CPU 0,0172 s.

- Wyniki bez optymalizacji są stabilne, ale pokazują większe wykorzystanie CPU.

Z optymalizacją:

- Średni czas zegarowy wzrósł nieznacznie do 0,5353 s, co oznacza wzrost o około 2% w stosunku do wersji bez optymalizacji.

- Średni czas CPU zmniejszył się do 0,0098 s, co stanowi znaczącą redukcję (o około 43%) w porównaniu z wersją bez optymalizacji.

Wnioski dla clone

- Nieznaczny wzrost czasu zegarowego: Optymalizacja nieznacznie zwiększyła czas zegarowy o 2%, co może wynikać z narzutu generowanego przez algorytmy optymalizacyjne.

- Duża oszczędność CPU: Optymalizacja zmniejszyła czas CPU o 43%, co wskazuje na znaczną efektywność optymalizacji, która może zmniejszyć obciążenie CPU przy tworzeniu wątków.

Porównanie fork i clone

- Tworzenie wątku (clone) zajmuje znacznie mniej czasu niż tworzenie procesu (fork). Narzut systemowy na tworzenie wątku jest znacznie mniejszy, co potwierdza tezę, że tworzenie procesów jest bardziej kosztowne pod względem zasobów systemowych niż tworzenie wątków.

- Zarówno dla fork, jak i clone, optymalizacja znacząco zmniejszyła zużycie czasu CPU, co jest najbardziej widoczne w przypadku clone, gdzie oszczędność wynosi 43%. To sugeruje, że wątki są bardziej korzystne przy zastosowaniu optymalizacji, jeśli priorytetem jest oszczędność zasobów CPU.

Analiza wyników dla program.c

W programie każdy wątek wykonuje 100,000 iteracji pętli, w której:

zmienna globalna (zmienna globalna) jest inkrementowana o 1,

zmienna lokalna (zmienna lokalna, przekazana jako argument) również jest inkrementowana o 1.

Wyniki bez optymalizacji

- Średni czas standardowy wyniósł 0,0001095 s, a średni czas CPU 0,0001123 s.

- Średni czas zegarowy wyniósł 0,002338 s, co oznacza, że wykonanie programu bez optymalizacji zajmuje niewiele czasu na poziomie standardowym i CPU, jednak czas zegarowy jest wyższy, ponieważ uwzględnia on czas potrzebny na operacje systemowe i synchronizację.

Wyniki z optymalizacją

- Średni czas standardowy wyniósł 0,0001243 s, a średni czas CPU 0,0001248 s.

- Średni czas zegarowy wyniósł 0,0023805 s, co oznacza niewielki wzrost o około 1,8% w stosunku do wersji bez optymalizacji.

Wnioski końcowe

- Przyjmując, że nowoczesne procesory wykonują miliardy operacji na sekundę (przykładowo 3 GHz procesor wykonuje do 3 miliardów operacji w ciągu sekundy), czas zegarowy dla clone wynoszący około **0,5 s** pozwala procesorowi wykonać teoretycznie około 1,5 miliarda operacji arytmetycznych w czasie, który zajmuje stworzenie wątku.

- Operacje wejścia/wyjścia są zazwyczaj znacznie wolniejsze niż operacje arytmetyczne. Jeśli założymy, że typowa operacja I/O trwa ok. 1 ms, procesor mógłby wykonać do 500 operacji wejścia/wyjścia w czasie potrzebnym na stworzenie pojedynczego wątku.

Wpływ optymalizacji na fork	Wpływ optymalizacji na clone
Optymalizacja lekko zmniejszyła czas zegarowy procesu fork (spadek o około 1,8%).	W przypadku clone czas zegarowy nieco wzrósł (około 2% wzrost), co może być efektem optymalizacji kodu, które uwzględniają redundancję operacji w pętli tworzenia wątków.
Przyczyna	
Różnice w czasie zegarowym mogą wynikać z eliminacji niektórych nieistotnych operacji w kodzie przez kompilator, co mogło mieć większy wpływ na fork niż na clone. Optymalizacja miała jednak minimalny wpływ na czas samego tworzenia wątków i procesów, co sugeruje, że proces tworzenia nowego kontekstu procesora (jak w fork) i kopiowanie przestrzeni pamięci są niezmiennie kosztowne.	

W program.c rozbieżności między zmiennymi lokalnymi a globalnymi wynikają z braku synchronizacji. Wątki równolegle modyfikują zmienną globalną i lokalne zmienne, przez co dochodzi do tzw. warunków wyścigu (race conditions). Oznacza to, że instrukcje modyfikujące wartości zmiennych są wykonywane przez oba wątki jednocześnie, co prowadzi do nieoczekiwanych wyników. Dzieje się tak, ponieważ inkrementacja zmiennej wymaga kilku operacji maszynowych (odczyt, zwiększenie, zapis), a wątki mogą „przeplatać" swoje operacje, powodując utratę niektórych inkrementacji.

Zmienne globalne:

- Jest współdzielona między wątkami, co oznacza, że każdy wątek może ją modyfikować.

- Brak synchronizacji powoduje, że ostateczna wartość jest niższa niż oczekiwana, ponieważ wątki wielokrotnie odczytują i modyfikują tę zmienną równocześnie. Przez to część operacji inkrementacji jest „gubiona".

Zmienne lokalne:

- Każdy wątek ma swoją kopię zmiennej lokalnej, co eliminuje bezpośrednie współdzielenie i w efekcie zmniejsza ryzyko kolizji.

- Mimo to, z powodu wspólnego dostępu do pamięci oraz braku synchronizacji, wartości lokalnych zmiennych różnią się od oczekiwanych. Obie zmienne lokalne wynoszą ostatecznie 192,069, zamiast 200,000, co sugeruje, że przy jednoczesnym zwiększaniu wartości zmiennych niektóre operacje zostały „utracone".

Podsumowanie

Brak synchronizacji w modyfikacji zmiennych globalnych prowadzi do utraty części operacji i niespójnych wyników.

Zmienne lokalne są mniej podatne na konflikty między wątkami, ale ich wartość końcowa również nie odpowiada dokładnie liczbie iteracji, co również jest efektem współbieżności bez zabezpieczeń.

Lab 3

Cel ćwiczenia

Nabycie umiejętności manipulacji wątkami przy użyciu biblioteki Pthreads, w tym tworzenia, niszczenia i synchronizacji wątków.

Przetestowanie mechanizmów przesyłania argumentów do wątków.

Zapoznanie się z funkcjonowaniem obiektów określających atrybuty wątków.

Wykonanie ćwiczeń

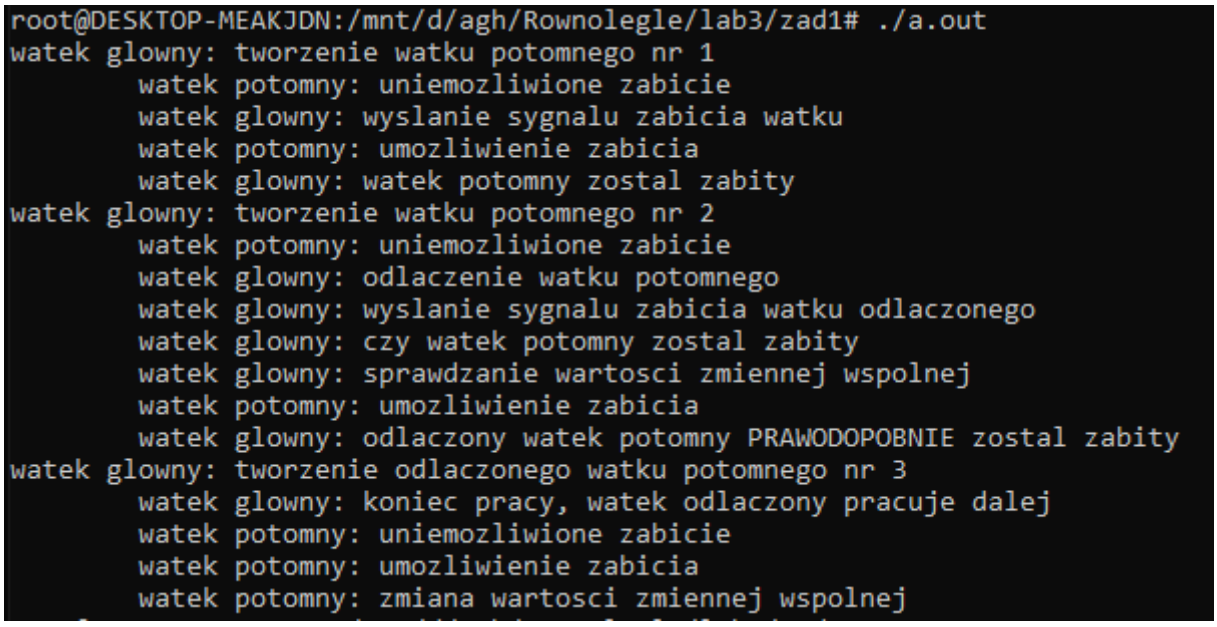
Modyfikacja i uruchomienie programu pthreads_detach_kill.c.

Napisanie i uruchomienie nowego programu w zad2/program.c do testowania przekazywania identyfikatorów do wątków.

Zadanie 1: Modyfikacja i analiza programu pthreads_detach_kill.c

Na początku uzupełniliśmy kod programu pthreads_detach_kill.c, dodając instrukcje dotyczące tworzenia i odłączania wątków oraz ustawiania ich stanów.

Wyniki działania i analiza



Wątki w Pthreads mogą funkcjonować w dwóch trybach:

- Przyłączalnym (joinable) – domyślny tryb, w którym główny wątek (lub inny wątek) może "czekać" na zakończenie tego wątku za pomocą pthread_join().

- Odłączonym (detached) – wątek, który działa niezależnie i nie można do niego dołączyć. Zasoby tego wątku są zwalniane automatycznie po jego zakończeniu.

Wątek przyłączalny wymaga, aby inny wątek dołączył się do niego (np. za pomocą pthread_join), co pozwala sprawdzić, czy zakończył swoje działanie, a także uzyskać wartość zwróconą.

Wątek odłączony jest automatycznie zwalniany po zakończeniu działania. Do takiego wątku nie można dołączyć, ponieważ jego zasoby są zarządzane automatycznie.

Wątek kończy swoje działanie, gdy wykona wszystkie instrukcje w swojej funkcji lub gdy napotka pthread_exit. Można również zakończyć go zewnętrznie za pomocą pthread_cancel.

Można wymusić zakończenie wątku za pomocą funkcji pthread_cancel. Należy jednak pamiętać, że wątek musi mieć ustawiony stan umożliwiający anulowanie (cancel state), aby pthread_cancel było skuteczne.

Wątek odłączony nie daje możliwości użycia pthread_join, więc nie można sprawdzić jego statusu po zakończeniu. W przypadku wymuszenia jego zakończenia nie mamy bezpośredniego dostępu do sprawdzenia statusu zakończenia.

Wątek przyłączalny umożliwia sprawdzenie, czy został zakończony przez pthread_join, dzięki czemu możemy zweryfikować, czy został "zabity" przez pthread_cancel.

Wątek może ustawić swój stan na PTHREAD_CANCEL_DISABLE za pomocą pthread_setcancelstate, co uniemożliwia jego "zabicie" przez pthread_cancel. Może też włączyć stan anulowania ponownie, gdy zakończy wykonywanie krytycznych operacji.

Aby sprawdzić, czy wątek został "zabity", można użyć pthread_join i sprawdzić wartość zwróconą przez wątek. Jeśli jest równa PTHREAD_CANCELED, oznacza to, że wątek został anulowany. W przypadku wątku odłączonego nie mamy bezpośredniego dostępu do takiego sprawdzenia, więc można ocenić to po zmianach w zmiennych globalnych lub innych zasobach.

Zadanie 2

W programie dla zadania 2 należy utworzyć i uruchomić kilka wątków, które wypisują swój systemowy identyfikator (wynik funkcji pthread_self()) oraz unikalny identyfikator przesyłany jako argument. Właściwe przesyłanie tego identyfikatora do wątku jest kluczowe, ponieważ błędy synchronizacji mogą prowadzić do sytuacji, gdzie dwa lub więcej wątków wyświetli ten sam identyfikator, co jest niepożądane.

Błąd synchronizacji przy użyciu wskaźnika do zmiennej

W pierwszej wersji programu użyto zmiennej identyfikator, która pełni rolę zmiennej sterującej w pętli tworzenia wątków. Adres tej samej zmiennej przekazywany jest do wszystkich tworzonych wątków, co powoduje, że wątki odczytują bieżącą wartość identyfikatora z tej samej lokalizacji pamięci. Ponieważ wątki działają równolegle z główną pętlą programu, wartość identyfikatora może zmienić się, zanim wątek zakończy swoją pracę i wypisze swoją wartość. To powoduje błędne wyświetlanie identyfikatorów wątków.

```
int identyfikator;

    // Tworzenie wątków
    for (int i = 0; i < liczba_watkow; i++) {
        identyfikator = i;
        if (pthread_create(&watki[i], NULL, zadanie_watku, &identyfikator) != 0) {
            perror("Błąd przy tworzeniu wątku");
            exit(1);
        }
    }
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab3/zad2# ./a.out
Wątek: ID systemowy: 140445081704000, Identyfikator wątku: 2
Wątek: ID systemowy: 140445073311296, Identyfikator wątku: 2
Wątek: ID systemowy: 140445048133184, Identyfikator wątku: 4
Wątek: ID systemowy: 140445056525888, Identyfikator wątku: 4
Wątek: ID systemowy: 140445064918592, Identyfikator wątku: 3
```

W powyższym przykładzie widać, że dwa wątki wypisały ten sam identyfikator (2 lub 4), co jest efektem błędu synchronizacji wynikającego z użycia wskaźnika do zmiennej identyfikator.

Aby rozwiązać problem, należy przypisać unikalną wartość identyfikatora każdemu wątkowi, przechowując je w osobnych miejscach w pamięci. W drugiej wersji programu, zamiast używać jednej zmiennej identyfikator, użyto tablicy identyfikatory, gdzie każdy element reprezentuje indywidualny identyfikator dla konkretnego wątku. Do funkcji pthread_create przekazywany jest wskaźnik do odpowiedniego elementu tej tablicy, dzięki czemu każdy wątek otrzymuje swój unikalny identyfikator.

```c
int identyfikatory[liczba_watkow];

    // Tworzenie wątków
    for (int i = 0; i < liczba_watkow; i++) {
        identyfikatory[i] = i;
        if (pthread_create(&watki[i], NULL, zadanie_watku, &identyfikatory[i]) != 0) {
            perror("Błąd przy tworzeniu wątku");
            exit(1);
        }
    }
```

```
Wątek: ID systemowy: 140357142431296, Identyfikator wątku: 0
Wątek: ID systemowy: 140357134038592, Identyfikator wątku: 1
Wątek: ID systemowy: 140357125645888, Identyfikator wątku: 2
Wątek: ID systemowy: 140357048202816, Identyfikator wątku: 3
Wątek: ID systemowy: 140357039810112, Identyfikator wątku: 4
```

Wnioski

Wątki odłączone i przyłączalne różnią się sposobem zarządzania zasobami. Wątek przyłączalny może być monitorowany, a jego zakończenie może być sprawdzone przez inny wątek. Wątek odłączony działa niezależnie, a jego zasoby są automatycznie zwalniane po zakończeniu.

Przesyłanie argumentów do wątków wymaga ostrożności – użycie jednej zmiennej wskaźnikowej w pętli może prowadzić do błędów synchronizacji i przypisania niepoprawnych wartości identyfikatorów do wątków.

Dla poprawności działania programu zaleca się stosowanie osobnych zmiennych lokalnych lub dynamicznej alokacji pamięci dla każdego argumentu przekazywanego do wątku.