|  |  |
| --- | --- |
| Laboratorium nr. 2, laboratorium nr. 3 | |
| Imię i nazwisko: Miłosz Dębowski | Kierunek: Informatyka Techniczna |
| Numer albumu: 415045 | Gr. Lab.: 8 |

**Laboratorium nr. 2**

1. Cel:

* przeprowadzenie pomiaru czasu CPU i zegarowego tworzenia procesów i wątków systemowych Linuxa
* nabycie umiejętności pisania programów wykorzystujących tworzenie wątków i procesów

1. Wykonanie

* Utworzenie folderów roboczych.
* Pobranie i rozpakowanie plików potrzebnych do realizacji ćwiczenia.
* Edycja kodu w pliku clone.c - załączenie biblioteki z folderu nadrzędnego, oraz dodanie wypisywania wartości zmiennej globalnej w funkcji funkcja\_watku():

#include  "pomiar\_czasu/pomiar\_czasu.h"

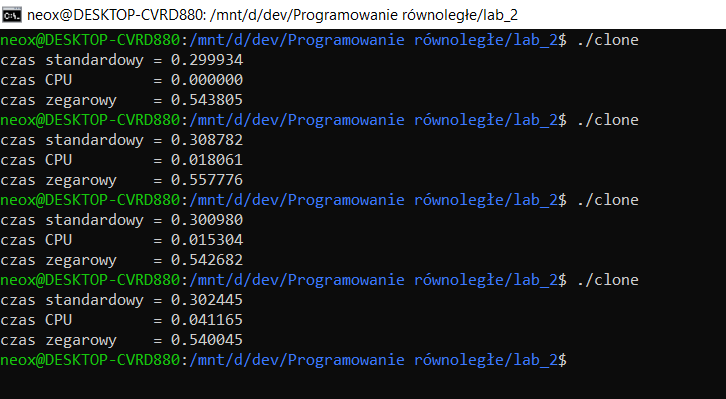
…

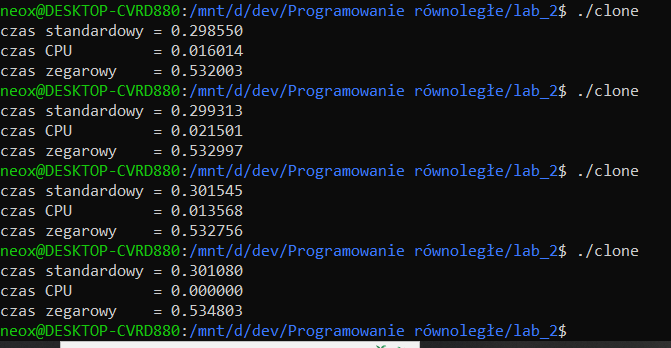
printf(„Zmienna globalna = %d\n”, zmienna\_globalna);

* Dodanie funkcji inicjuj\_czas() oraz drukuj\_czas(), kolejno przed i po pętli wykonującej obliczenia.
* Analogiczne wykonanie zmian w kodzie programu fork.c
* Napisanie nowego programu na podstawie programu clone.c w

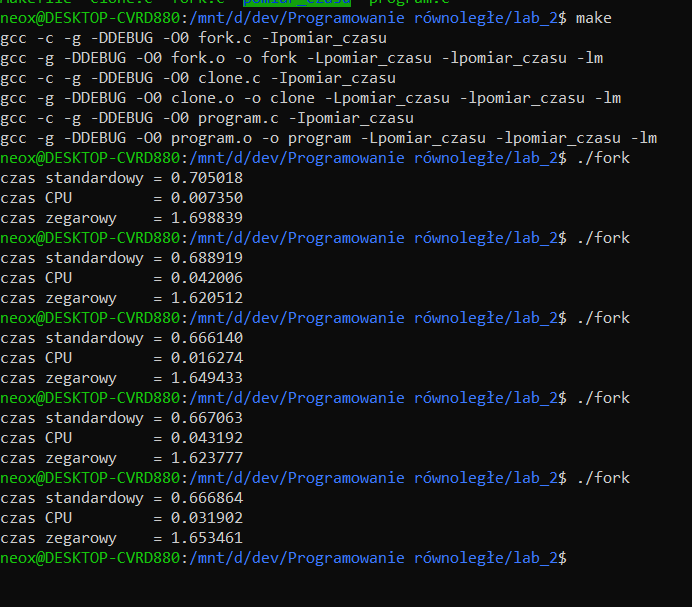
którym, bezpośrednio po sobie tworzone są dwa wątki do działania równoległego. Wątki zwiększają w odpowiednio długiej pętli (co najmniej 100000 iteracji) wartości dwóch zmiennych - jedną globalną a drugą lokalną.

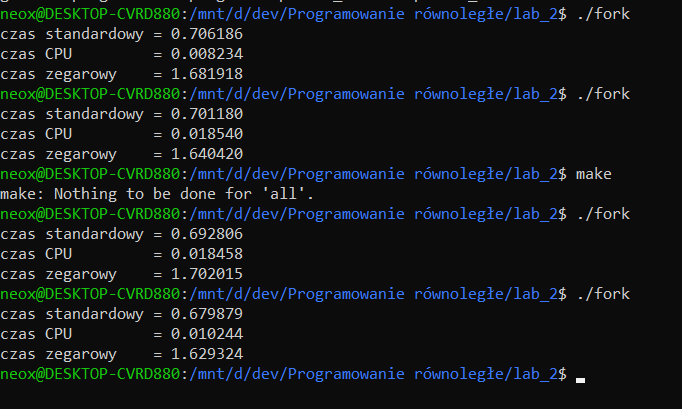
1. Otrzymane wyniki

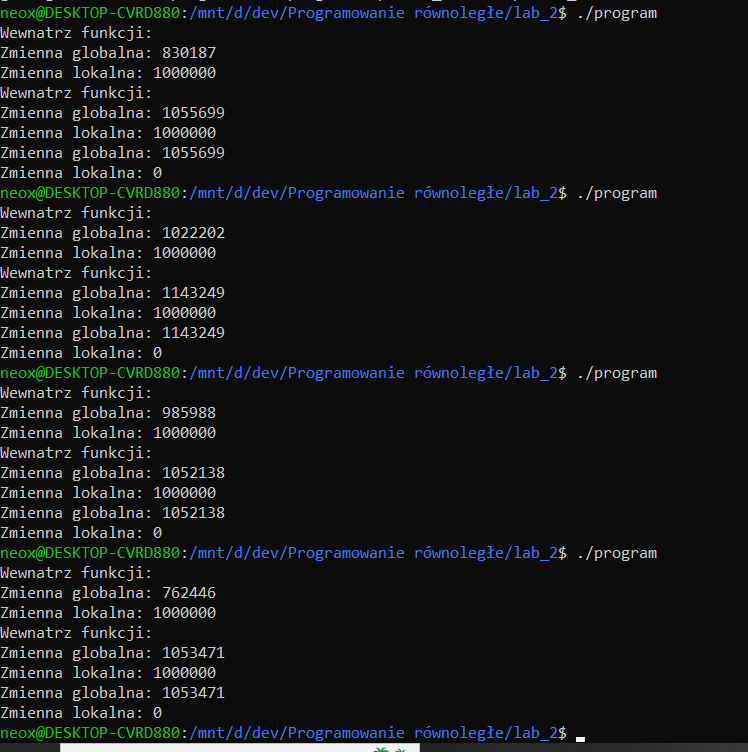
Clone – wersja do debugowania

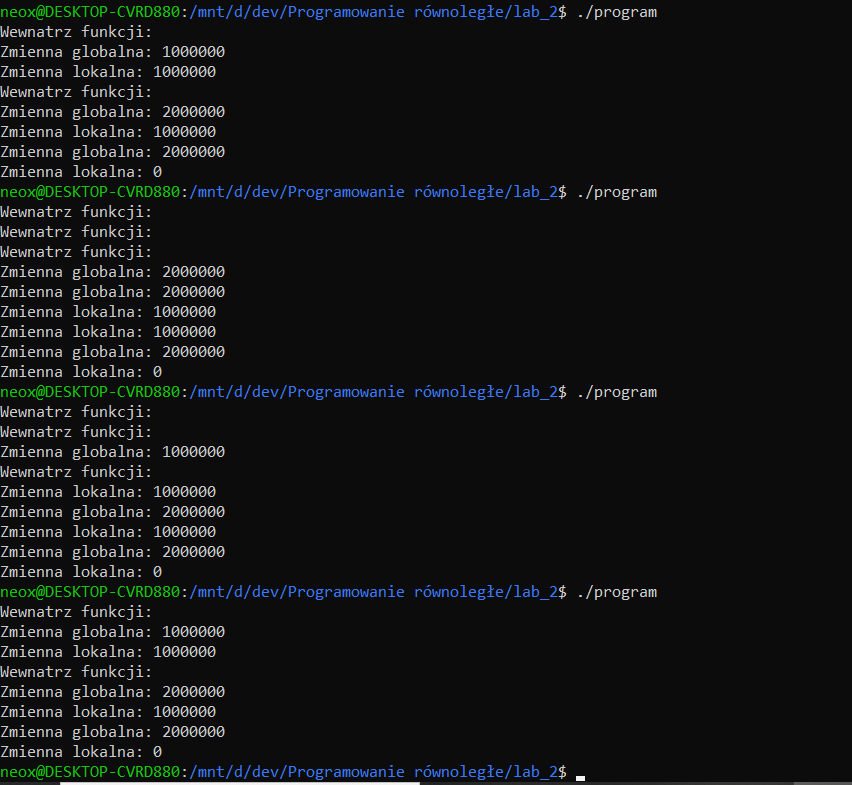


Clone – wersja zoptymalizowana

Fork – wersja do debugowania

Fork – wersja zoptymalizowana

Program – wersja do debugowania

Program – wersja zoptymalizowana

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Wersja DDEBUG | Czas tworzenia wątku (clone) | | | Czas tworzenia procesu (fork) | | |
| czas stnadardowy (s) | czas cpu (s) | czas zegarowy (s) | czas stnadardowy (s) | czas cpu (s) | czas zegarowy (s) |
| Pomiar 1 | 0,3087820000 | 0,0180610000 | 0,5577760000 | 0,7050180000 | 0,00735 | 1,698839 |
| Pomiar 2 | 0,3009800000 | 0,0153040000 | 0,5426820000 | 0,6889190000 | 0,042006 | 1,620512 |
| Pomiar 3 | 0,3024450000 | 0,0411650000 | 0,5400450000 | 0,6661400000 | 0,016274 | 1,649433 |
| Uśredniony wynik | 0,3040690000 | 0,0248433333 | 0,5468343333 | 0,6866923333 | 0,02187667 | 1,656261333 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Wersja -O3 | Czas tworzenia wątku (clone) | | | Czas tworzenia procesu (fork) | | |
| czas stnadardowy (s) | czas cpu (s) | czas zegarowy (s) | czas stnadardowy (s) | czas cpu (s) | czas zegarowy (s) |
| Pomiar 1 | 0,29855 | 0,016014 | 0,532003 | 0,706186 | 0,008234 | 1,681918 |
| Pomiar 2 | 0,299313 | 0,021501 | 0,532997 | 0,70118 | 0,01854 | 1,64042 |
| Pomiar 3 | 0,301545 | 0,013568 | 0,532756 | 0,692806 | 0,018458 | 1,702015 |
| Uśredniony wynik | 0,299802667 | 0,017027667 | 0,532585333 | 0,700057333 | 0,01507733 | 1,674784333 |

1. Kod programów

clone.c

1. #define \_GNU\_SOURCE
2. #include<stdlib.h>
3. #include<stdio.h>
4. #include<unistd.h>
5. #include "pomiar\_czasu/pomiar\_czasu.h"
6. #include <sys/types.h>
7. #include <sys/wait.h>
8. #include <sched.h>
9. #include <linux/sched.h>
10. int zmienna\_globalna=0;
11. #define ROZMIAR\_STOSU 1024\*64
12. int funkcja\_watku( void\* argument )
13. {
14. zmienna\_globalna++;
15. return 0;
16. }
17. int main()
18. {
19. void \*stos;
20. pid\_t pid;
21. int i;
22. inicjuj\_czas();
23. stos = malloc( ROZMIAR\_STOSU );
24. if (stos == 0) {
25. printf("Proces nadrzędny - blad alokacji stosu\n");
26. exit( 1 );
27. }
28. for(i=0;i<10000;i++){
29. pid = clone( &funkcja\_watku, (void \*) stos+ROZMIAR\_STOSU,
30. CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND | CLONE\_VM, 0 );
31. waitpid(pid, NULL, \_\_WCLONE);
32. }
33. drukuj\_czas();
34. free( stos );
35. }

fork.c

1. #include<stdlib.h>
2. #include<stdio.h>
3. #include<unistd.h>
4. #include <sys/types.h>
5. #include <sys/wait.h>
6. #include"pomiar\_czasu/pomiar\_czasu.h"
7. int zmienna\_globalna=0;
8. int main(){
9. int pid, wynik, i;
10. inicjuj\_czas();
11. for(i=0;i<10000;i++){
12. pid = fork();
13. if(pid==0){
14. zmienna\_globalna++;
15. // char arg1[] = "/bin/ls";
16. // char arg2[] = ".";
17. // char\* arg[] = {arg1,arg2,NULL};
18. // char\* arg[] = {"/bin/ls",".",NULL};
19. // wynik=execv("/bin/ls",arg);
20. // wynik=execv("./program",NULL);
21. // if(wynik==-1)
22. //      printf("Proces potomny nie wykonal programu\n");
23. exit(0);
24. } else {
25. wait(NULL);
26. }
27. }
28. drukuj\_czas();
29. }

program.c

1. #define \_GNU\_SOURCE
2. #include<stdlib.h>
3. #include<stdio.h>
4. #include<unistd.h>
5. #include "pomiar\_czasu/pomiar\_czasu.h"
6. #include <sys/types.h>
7. #include <sys/wait.h>
8. #include <sched.h>
9. #include <linux/sched.h>
10. int zmienna\_globalna=0;
11. #define ROZMIAR\_STOSU 1024\*64
12. int funkcja\_watku( void\* argument )
13. {
14. int zmienna\_lokalna = \*(int \*)argument;
15. for(int i = 0; i < 1e+8; i++){
16. zmienna\_globalna++;
17. zmienna\_lokalna++;
18. }
19. printf("Wewnatrz funkcji:\n");
20. printf("Zmienna globalna: %d\n",zmienna\_globalna);
21. printf("Zmienna lokalna: %d\n",zmienna\_lokalna);
22. return 0;
23. }
24. int main()
25. {
26. void \*stos,\*stos1;
27. pid\_t pid,pid1;
28. int i;
29. int zmienna\_lokalna = 0;
30. stos = malloc( ROZMIAR\_STOSU );
31. stos1 = malloc( ROZMIAR\_STOSU );
32. if (stos == 0) {
33. printf("Proces nadrzędny - blad alokacji stosu\n");
34. exit( 1 );
35. }
36. pid = clone( &funkcja\_watku, (void \*) stos+ROZMIAR\_STOSU,CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND | CLONE\_VM, &zmienna\_lokalna );
37. pid1 = clone( &funkcja\_watku, (void \*) stos1+ROZMIAR\_STOSU, CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND | CLONE\_VM, &zmienna\_lokalna );
38. waitpid(pid, NULL, \_\_WCLONE);
39. waitpid(pid1, NULL, \_\_WCLONE);
40. printf("Zmienna globalna: %d\n",zmienna\_globalna);
41. printf("Zmienna lokalna: %d\n",zmienna\_lokalna);
42. free( stos );
43. free(stos1);
44. }
45. Wnioski

* Wykonywanie zadań przez wątki jest około dwa razy szybsze niż w przypadku procesów, gdy zadania są identyczne.
* Funkcja fork() to podstawowa metoda tworzenia procesów w systemach Unix, która dzieli proces wywołujący na dwa: proces "rodzica" i proces "dziecka". Różnica między nimi polega na wartości zwracanej przez fork() - proces rodzica otrzymuje PID procesu dziecka, proces dziecka - wartość 0, a w przypadku błędu proces potomny nie powstaje, a funkcja zwraca -1.
* Funkcja clone() jest rozbudowaną wersją funkcji fork(), która umożliwia bardziej szczegółowe kontrolowanie procesu tworzenia nowych procesów i pozwala na współdzielenie zasobów między procesami.
* Zoptymalizowana wersja programu.c zwraca wartość zmiennej globalnej dwa razy większą niż lokalnej. Wynika to z optymalizacji polegającej na jednorazowym dodaniu 10000 do zmiennej lokalnej i 20000 do globalnej zamiast dodawania +1 w każdej z 10000 operacji. Takie działanie uznaje się za nadmiernie agresywną optymalizację.

**Laboratorium nr. 3**

1. Cel:

* nabycie praktycznej umiejętności manipulowania wątkami Pthreads – tworzenia, niszczenia, elementarnej synchronizacji
* przetestowanie mechanizmu przesyłania argumentów do wątku
* poznanie funkcjonowania obiektów określających atrybuty wątków.

1. Wykonanie

* Utworzenie folderów roboczych.
* Pobranie i rozpakowanie plików potrzebnych do realizacji ćwiczenia.
* Edycja kodu w pliku pthreads\_detach\_kill.c - uzupełnienie kodu programu w opraciu o skrypt podany przez profesora
* Utworzenie pliku Makefile, uruchomienie i przetestowanie programu
* Zaprojektowanie, utworzenie i przetestowanie nowej procedury wątków, do której jako argument przesyłany jest identyfikator każdego wątku, z zakresu 0..liczba\_wątków-1

1. Obraz zawierający tekst, zrzut ekranu, Czcionka

   Opis wygenerowany automatycznieOtrzymane wyniki

pthreads\_detach\_kill.c – zadanie 1

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatyczniezadanie 2

1. Kod programów

Makefile – zadanie 1

1. # kompilator c
2. CCOMP = gcc
3. # konsolidator
4. LOADER = gcc
5. # opcje optymalizacji:
6. # wersja do debugowania
7. OPT = -g -DDEBUG
8. # wersja zoptymalizowana do mierzenia czasu
9. # OPT = -O3
10. # zaleznosci i komendy
11. zad1: pthreads\_detach\_kill.o
12. $(LOADER) $(OPT) pthreads\_detach\_kill.o -o zad1 -lpthread $(LIB)
13. # jak uzyskac plik pthreads\_kill\_detach.o ?
14. pthreads\_detach\_kill.o: pthreads\_detach\_kill.c
15. $(CCOMP) -c $(OPT) pthreads\_detach\_kill.c
16. clean:
17. rm -f \*.o zad1

Makefile – zadanie 2

1. # kompilator c
2. CCOMP = gcc
3. # konsolidator
4. LOADER = gcc
5. # opcje optymalizacji:
6. # wersja do debugowania
7. OPT = -g -DDEBUG
8. # wersja zoptymalizowana do mierzenia czasu
9. # OPT = -O3
10. # zaleznosci i komendy
11. zad2: zad2.o
12. $(LOADER) $(OPT) zad2.o -o zad2 -lpthread $(LIB)
13. # jak uzyskac plik pthreads\_kill\_detach.o ?
14. zad2.o: zad2.c
15. $(CCOMP) -c $(OPT) zad2.c
16. clean:
17. rm -f \*.o zad2

pthreads\_detach\_kill.c

1. #include<stdlib.h>
2. #include<stdio.h>
3. #include<pthread.h>
4. #include <unistd.h>
5. int zmienna\_wspolna=0;
6. #define WYMIAR 1000
7. #define ROZMIAR WYMIAR\*WYMIAR
8. double a[ROZMIAR],b[ROZMIAR],c[ROZMIAR];
9. double czasozajmowacz(){
10. int i, j, k;
11. int n=WYMIAR;
12. for(i=0;i<ROZMIAR;i++) a[i]=1.0\*i;
13. for(i=0;i<ROZMIAR;i++) b[i]=1.0\*(ROZMIAR-i);
14. for(i=0;i<n;i++){
15. for(j=0;j<n;j++){
16. c[i+n\*j]=0.0;
17. for(k=0;k<n;k++){
18. c[i+n\*j] += a[i+n\*k]\*b[k+n\*j];
19. }
20. }
21. }
22. return(c[ROZMIAR-1]);
23. }
24. void \* zadanie\_watku (void \* arg\_wsk)
25. {
26. pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE, NULL);
27. printf("\twatek potomny: uniemozliwione zabicie\n");
28. czasozajmowacz();
29. printf("\twatek potomny: umozliwienie zabicia\n");
30. pthread\_setcancelstate(PTHREAD\_CANCEL\_ENABLE, NULL);
31. pthread\_testcancel()
32. zmienna\_wspolna++;
33. printf("\twatek potomny: zmiana wartosci zmiennej wspolnej\n");
34. return(NULL);
35. }
36. int main()
37. {
38. pthread\_t tid;
39. pthread\_attr\_t attr;
40. void \*wynik;
41. int i;
42. //Wątek przyłączalny
43. printf("watek glowny: tworzenie watku potomnego nr 1\n");
44. /\*Tu wstaw kod tworzenia wątku z domyślnymi własnościami\*/
45. pthread\_create(&tid, NULL, zadanie\_watku, NULL);//dopisane
46. sleep(2); // czas na uruchomienie watku
47. printf("\twatek glowny: wyslanie sygnalu zabicia watku\n");
48. pthread\_cancel(tid);
49. //Co nalezy zrobić przed sprawdzeniem czy wątki się skonczyły?
50. pthread\_join(tid, &wynik);//dopisane
51. if (wynik == PTHREAD\_CANCELED)
52. printf("\twatek glowny: watek potomny zostal zabity\n");
53. else
54. printf("\twatek glowny: watek potomny NIE zostal zabity - blad\n");
55. //Odłączanie wątku
56. zmienna\_wspolna = 0;
57. printf("watek glowny: tworzenie watku potomnego nr 2\n");
58. /\*Tu wstaw kod tworzenia wątku z domyślnymi własnościami\*/
59. pthread\_create(&tid, NULL, zadanie\_watku, NULL);//dopisane
60. sleep(2); // czas na uruchomienie watku
61. printf("\twatek glowny: odlaczenie watku potomnego\n");
62. //Instrukcja odłączenia?
63. pthread\_detach(tid);//dopisane
64. printf("\twatek glowny: wyslanie sygnalu zabicia watku odlaczonego\n");
65. pthread\_cancel(tid);
66. //Czy watek został zabity? Jak to sprawdzić?
67. printf("\twatek glowny: czy watek potomny zostal zabity \n");//całe dopisane
68. printf("\twatek glowny: sprawdzanie wartosci zmiennej wspolnej\n");
69. for(i=0;i<10;i++)
70. {
71. sleep(1);
72. if(zmienna\_wspolna!=0) break;
73. }
74. if (zmienna\_wspolna==0) printf("\twatek glowny: odlaczony watek potomny PRAWDOPODOBNIE zostal zabity\n");
75. else printf("\twatek glowny: odlaczony watek potomny PRAWDOPODOBNIE NIE zostal zabity\n");
76. //Wątek odłączony
77. //Inicjacja atrybutów?
78. pthread\_attr\_init(&attr);//dopisane
79. //Ustawianie typu watku na odłaczony
80. pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);//dopisane
81. printf("watek glowny: tworzenie odlaczonego watku potomnego nr 3\n");
82. pthread\_create(&tid, &attr, zadanie\_watku, NULL);
83. //Niszczenie atrybutów
84. pthread\_attr\_destroy(&attr);//dopisane
85. printf("\twatek glowny: koniec pracy, watek odlaczony pracuje dalej\n");
86. pthread\_exit(NULL); // co stanie sie gdy uzyjemy exit(0)?
87. }

zad2.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define THREADS\_COUNT 6

void\* zadanie\_watku(void\* arg\_wsk) {

    int moj\_arg = \*((int\*)arg\_wsk);

    printf("SYSTEM ID: %d & SELF ID = %ld\n", moj\_arg, pthread\_self());

    return NULL;

}

int main() {

    pthread\_t threads[THREADS\_COUNT];

    int index[THREADS\_COUNT];

    for (int i = 0; i < THREADS\_COUNT; i++) {

        index[i] = i;

        pthread\_create(&threads[i], NULL, zadanie\_watku, (void\*)&index[i]);

    }

    for (int i = 0; i < THREADS\_COUNT; i++) {

        pthread\_join(threads[i], NULL);

    }

    pthread\_exit(NULL);

}

1. Wnioski

* Program pthreads\_detach\_kill.c demonstruje różne metody zarządzania wątkami w języku C, w tym tworzenie, anulowanie i odłączanie wątków.
* Możliwość anulacji wątku można kontrolować, co jest istotne, gdy wątek wykonuje długotrwałe operacje i nie powinien być przerywany w trakcie ich wykonywania.
* Wątki mogą być przyłączalne lub odłączalne, co wpływa na sposób ich zarządzania oraz na to, jak i kiedy można sprawdzić ich zakończenie.
* Użycie funkcji sleep() w programie ilustruje sposób, w jaki można wprowadzić opóźnienia w działaniu wątków, co może być istotne w kontekście synchronizacji lub testowania.
* Oto wnioski płynące z podanego kodu i wyników:
* Każdy wątek ma swój unikalny SELF ID, co wskazuje, że są to niezależne wątki systemowe. SELF ID to identyfikator przypisywany przez system do każdego wątku, co potwierdza, że wątki są traktowane jako oddzielne jednostki wykonawcze.
* Mimo że wątki są tworzone w określonej kolejności, system operacyjny decyduje o ich rzeczywistym czasie rozpoczęcia i zakończenia, co może skutkować różnymi kolejnościami wypisywania wyników przy każdym uruchomieniu programu.