Programowanie równoległe - Laboratorium 2	
18.10.2024 r.	Szymon Figura Informatyka Techniczna, gr. lab. 2

Celem laboratoriów było nabycie umiejętności tworzenia wątków i procesów w systemach Linux przy użyciu funkcji fork oraz clone.

Po pobraniu i rozpakowaniu plików fork.c i clone.c oraz biblioteki służącej do pomiaru czasu, dodałem pliki odpowiedzialne za pomiar czasu pochodzące z biblioteki pomiar_czasu.c.

fork.c

```c
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 );
  }

  inicjuj_czas();
  for(i=0;i<1000;i++){

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

  }
  drukuj_czas();

  free( stos );
}
```

clone.c

```c
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 );
  }

  inicjuj_czas();
  for(i=0;i<1000;i++){

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

  }
  drukuj_czas();

  free( stos );
}
```

Następnie wykonałem po 3 pomiary czasu tworzenia się 1000 procesów i wątków w wersji z optymalizacją (-O3) oraz bez optymalizacji (-g -DDEBUG).

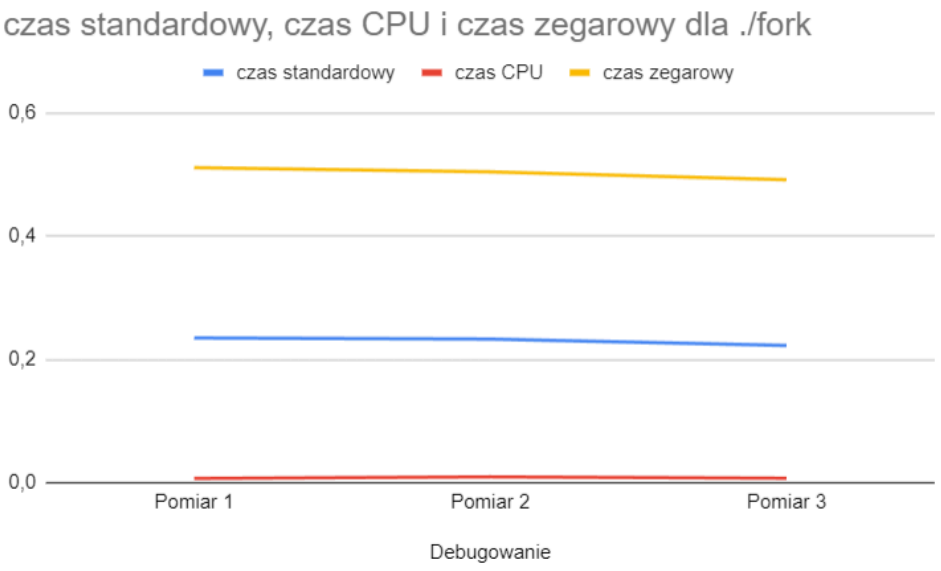
Debugowanie	-g -DDEBUG	./fork	
	Operacje wejścia/wyjścia		
	czas standardowy	czas CPU	czas zegarowy
Pomiar 1	0,234826	0,006839	0,511217
Pomiar 2	0,233017	0,009415	0,504352
Pomiar 3	0,222659	0,007175	0,491581
Średnia	0,2301673333	0,00780966666	0,5023833333

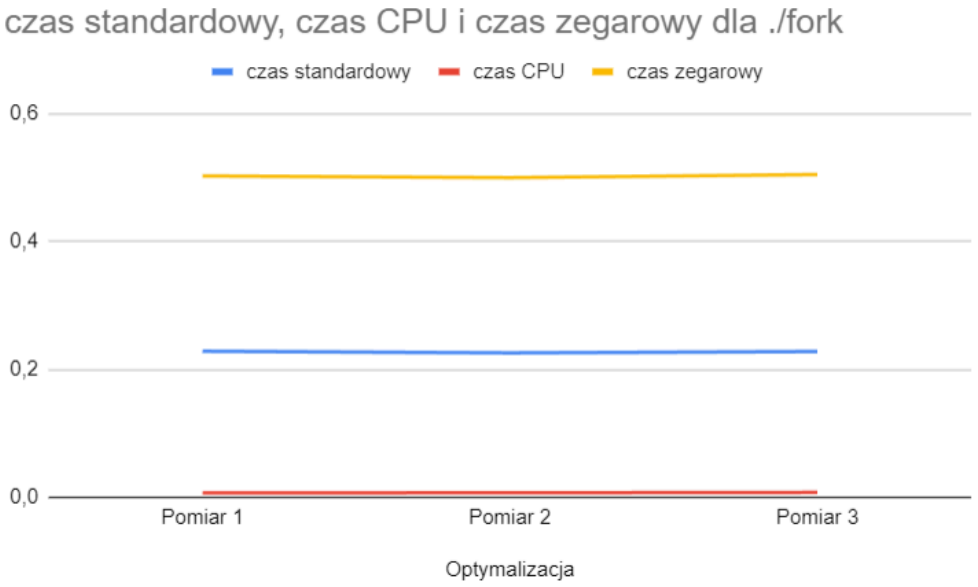
Optymalizacja	-O3	./fork	
	Operacje wejścia/wyjścia		
	czas standardowy	czas CPU	czas zegarowy
Pomiar 1	0,228333	0,006835	0,502458
Pomiar 2	0,225771	0,007068	0,499748
Pomiar 3	0,22787	0,007555	0,50462
Średnia	0,2273246667	0,00715266666	0,5022753333

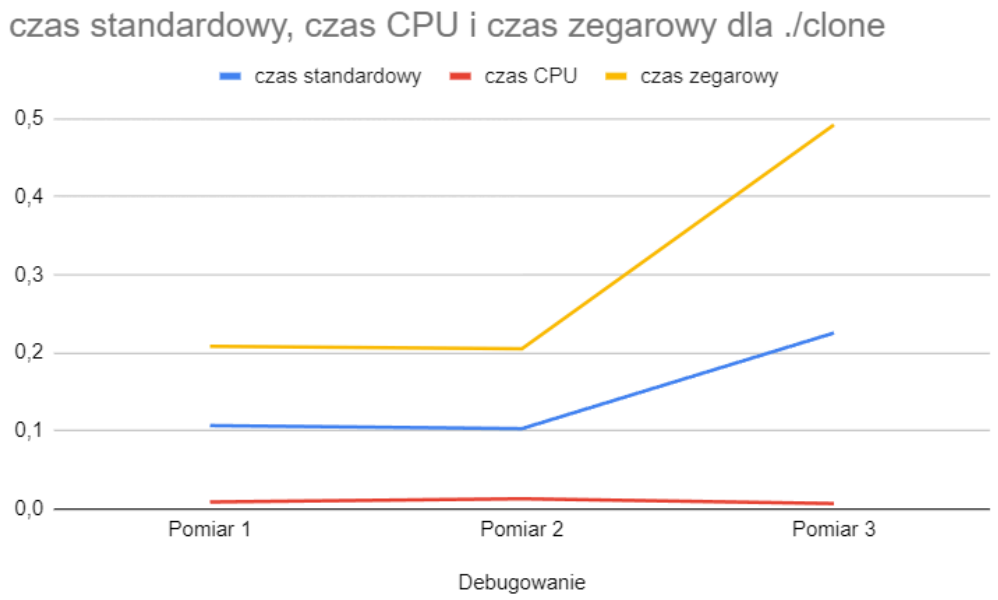
Debugowanie	-g -DDEBUG	./clone	
	Operacje wejścia/wyjścia		
	czas standardowy	czas CPU	czas zegarowy
Pomiar 1	0,106924	0,009052	0,208395
Pomiar 2	0,103004	0,013183	0,205376
Pomiar 3	0,22555	0,007048	0,492166
Średnia	0,1451593333	0,009761	0,301979

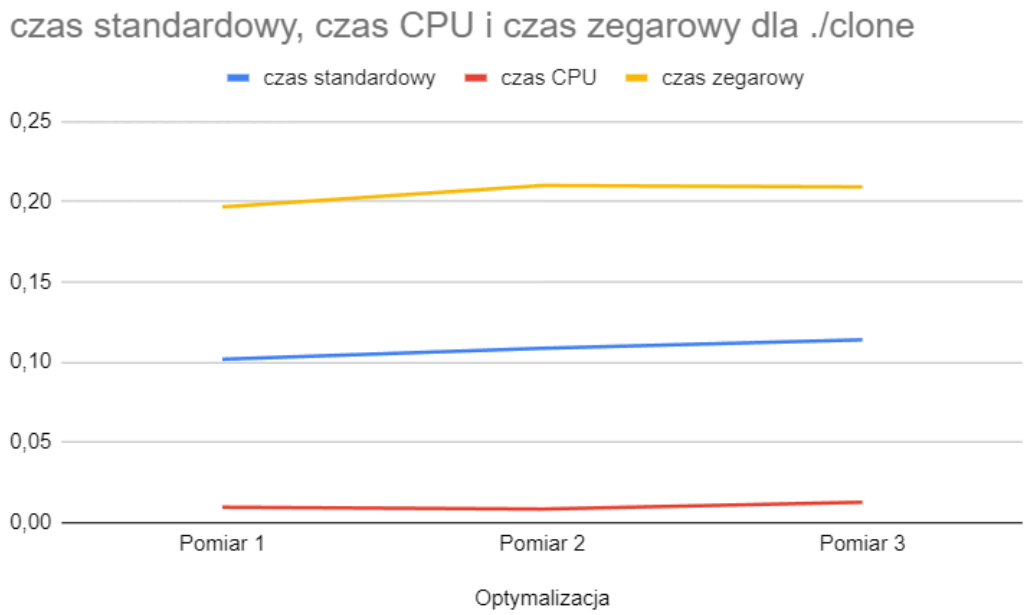
Optymalizacja	-O3	./clone	
	Operacje wejścia/wyjścia		
	czas standardowy	czas CPU	czas zegarowy
Pomiar 1	0,101883	0,009663	0,196834
Pomiar 2	0,108721	0,008496	0,210204
Pomiar 3	0,114014	0,01276	0,209314
Średnia	0,108206	0,01030633333	0,2054506667

Wykresy przedstawiające różnice w czasach









Następnie stworzyłem program new_clone.c bazując na clone.c, w którym tworzę dwa wątki. Ich zadaniem jest inkrementowanie zmiennej lokalnej oraz globalnej.

```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 )
{
  int zmienna_lokalna = 0;
  for(int i = 0; i < 100000; i++) {
    zmienna_lokalna++;
    zmienna_globalna++;
  }
  return 0;
}

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

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

  stos2 = malloc( ROZMIAR_STOSU );
  if (stos2 == 0) {
    printf("Proces nadrzędny - blad alokacji stosu\n");
    exit( 1 );
  }

  pid1 = clone( &funkcja_watku, (void *) stos+ROZMIAR_STOSU,
        CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, 0 );

  pid2 = clone( &funkcja_watku, (void *) stos2+ROZMIAR_STOSU,
        CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, 0 );
  waitpid(pid1, NULL, __WCLONE);
  waitpid(pid2, NULL, __WCLONE);

  printf("Zmienna globalna: %d\n", zmienna_globalna);

  free( stos );
}
```

Wnioski:

Funkcja fork() jest jedną z podstawowych metod tworzenia nowych procesów w systemach Unix. Kiedy program wywołuje tę funkcję, tworzy się nowy proces, czyli program dzieli się na dwa – macierzysty i potomny. Różnią się one tym, że funkcja fork() zwraca różne wartości: proces macierzysty dostaje numer PID nowego procesu, a proces potomny dostaje wartość 0. Funkcja clone() jest bardziej zaawansowaną wersją fork(). Daje większą kontrolę nad tym, które zasoby są współdzielone między procesami, co jest szczególnie ważne w programach działających równolegle. Dzięki clone(), nowy proces może korzystać z tej samej pamięci, co poprawia wydajność.

Zoptymalizowana wersja programu działała szybciej niż ta bez optymalizacji. Procesy tworzą się wolniej niż wątki. Szacunkowa liczba operacji arytmetycznych, które procesor mógłby wykonać w czasie tworzenia wątków, była ponad dwa razy większa w porównaniu do procesów. Z tego wynika, że wątki są bardziej efektywne pod względem użycia zasobów CPU.

Program new_clone.c, który tworzy dwa wątki, miał na celu inkrementowanie zmiennych lokalnych oraz globalnych. Zmienna globalna była modyfikowana poprawnie przez oba wątki, co wskazuje na współdzielenie pamięci między wątkami.