Imię i Nazwisko	Kierunek	Rok studiów i grupa
Patryk Twardosz	Informatyka Techniczna	I rok, Gr. 9
Data zajęć	**Numer i temat sprawozdania**	
14.10.2024r.	Lab 2 i 3 – Pomiar czasu clone/fork i POSIX	

Lab2:

Cel:

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

fork() - Tworzy nowy proces jako kopię procesu rodzica.

clone() - Bardziej elastyczna funkcja, umożliwiająca większą kontrolę nad współdzielonymi zasobami.

Zadanie:

1. Przygotowanie projektu
2. Uzupełnienie plików źródłowych o procedury pomiaru czasu.

- clone.c:

```
inicjuj_czas();  // rozpoczęcie pomiaru

for(i=0;i<1000;i++) {
   /* istniejąca już zawartość pętli */
}
printf("Zmienna globalna: %d\n",zmienna_globalna);
drukuj_czas(); // zakończenie pomiaru
```

- fork.c

```
inicjuj_czas();
for (i = 0; i < 1000; i++) {
   /* istniejąca już zawartość pętli */
}
printf("Zmienna globalna: %d\n", zmienna_globalna);
drukuj_czas();
```

3. Przeprowadzenie eksperymentu mierzącego czas tworzenie procesów i wątków.

Debug

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./clone
Zmienna globalna: 100000000
czas standardowy = 0.033251
czas CPU         = 0.000000
czas zegarowy    = 0.099144
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./clone
Zmienna globalna: 100000000
czas standardowy = 0.035665
czas CPU         = 0.000000
czas zegarowy    = 0.100753
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./clone
Zmienna globalna: 100000000
czas standardowy = 0.032425
czas CPU         = 0.000000
czas zegarowy    = 0.098872
```

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./fork
Zmienna globalna: 0
czas standardowy = 0.072202
czas CPU         = 0.000000
czas zegarowy    = 2.082233
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./fork
Zmienna globalna: 0
czas standardowy = 0.073879
czas CPU         = 0.000000
czas zegarowy    = 2.856627
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./fork
Zmienna globalna: 0
czas standardowy = 0.075648
czas CPU         = 0.007047
czas zegarowy    = 2.123530
```

Zoptymalizowane

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./clone
Zmienna globalna: 100000000
czas standardowy = 0.034994
czas CPU         = 0.027744
czas zegarowy    = 0.055921
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./clone
Zmienna globalna: 100000000
czas standardowy = 0.031763
czas CPU         = 0.000000
czas zegarowy    = 0.051790
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./clone
Zmienna globalna: 100000000
czas standardowy = 0.032340
czas CPU         = 0.000000
czas zegarowy    = 0.052279
```

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./fork
Zmienna globalna: 0
czas standardowy = 0.072763
czas CPU         = 0.014091
czas zegarowy    = 2.010611
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./fork
Zmienna globalna: 0
czas standardowy = 0.070698
czas CPU         = 0.000000
czas zegarowy    = 1.979254
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./fork
Zmienna globalna: 0
czas standardowy = 0.072634
czas CPU         = 0.000000
czas zegarowy    = 2.010334
```

4. Napisanie nowego programu tworzącego 2 wątki jednocześnie.

```
static int zmienna_globalna = 0;

int funkcja_watku(void *argument) {
  int *var = (int *)argument;
  for (int i = 0; i < 100000; i++) {
    zmienna_globalna++;
    (*var)++;
  }
  return 0;
}

int main() {
  /* deklaracja zmiennych I alokacja stosów */
  pid1 = clone(funkcja_watku, (void *) stos1 + ROZMIAR_STOSU, CLONE_FS | CLONE_FILES |
CLONE_SIGHAND | CLONE_VM, &local1);
  pid2 = clone(funkcja_watku, (void *) stos2 + ROZMIAR_STOSU, CLONE_FS | CLONE_FILES |
CLONE_SIGHAND | CLONE_VM, &local2);
  /* poczekanie na wątki I wyczyszczenie stosów */
  printf("Zmienna globalna: %d\n", zmienna_globalna);
  printf("Local1: %d\n", local1);
  printf("Local2: %d\n", local2);
}
```

Debug

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./test
Zmienna globalna: 155700
Local1: 100000
Local2: 100000
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./test
Zmienna globalna: 119011
Local1: 100000
Local2: 100000
```

Zoptymalizowane

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab2$ ./test
Zmienna globalna: 200000
Local1: 100000
Local2: 100000
```

5. Napisanie programu uruchamiającego inny program z przekazaniem argumentów.

Modyfikacja w fork.c:

```
char *args[] = {"./program", "Patryk", "Twardosz", NULL};
wynik = execv(args[0], args);
```

program.c:

```
int main(int argc, char *argv[]) {
  if(argc > 2)
    printf("With args: PID: %d, %s %s\n", getpid(), argv[1], argv[2]);
  else
    printf("Without args: PID: %d\n", getpid());
  return 0;
}
```

Wnioski:

- W przypadku fork() i clone() różnica w poziomie optymalizacji jest marginalna, dlatego że większość czasu programu spędzana jest na oczekiwaniu na system operacyjny, aby ten stworzył procesy/wątki.
- W przypadku modyfikacji przez 2 wątki tej samej zmiennej może dojść do korupcji danych, co widać w wersji niezoptymalizowanej, natomiast optymalizacje zmieniają pozbywają się pętli, więc automatycznie jest dużo mniejsza szansa na wykonanie operacji dodawania w tym samym czasie, jednak korupcja danych w ciąż jest możliwa.

Odpowiedzi na pytania:

- Tworzenie wątków jest 10-30 razy szybsze niż tworzenie procesów, ponieważ wątki współdzielą przestrzeń adresową z procesem-matką. W czasie potrzebnym na stworzenie wątku (np. 500 mikrosekund) procesor może wykonać ok. 1,5 miliona operacji arytmetycznych lub kilkadziesiąt do kilkuset operacji I/O, zależnie od typu nośnika. Optymalizacja kodu użytkownika ma niewielki wpływ na czas tworzenia procesów i wątków, ponieważ te operacje zależą od systemu operacyjnego.
- Aby sprawdzić poprawność operacji wątków, należy porównać oczekiwane wartości zmiennych po ich wykonaniu z wartościami rzeczywistymi – różnice mogą wynikać z braku

synchronizacji (race condition). Zmienne globalne i statyczne są współdzielone między wątkami, więc bez synchronizacji mogą prowadzić do niespójności. Zmienne lokalne są oddzielne dla każdego wątku, co zapobiega konfliktom między nimi.

- Rozmiar stosu dla nowo tworzonego wątku można określić za pomocą atrybutów wątku (pthread_attr_t) i funkcji pthread_attr_setstacksize(). Przekroczenie tego rozmiaru – np. poprzez rekurencję lub alokację dużych zmiennych lokalnych – powoduje *stack overflow* i może prowadzić do błędu segmentacji. Domyślny rozmiar stosu w wątkach w systemach Linux to zazwyczaj 8 MB, choć może się różnić w zależności od konfiguracji systemu.

Lab3:

Cel:

- Poznanie tworzenia, niszczenia i elementarnej synchronizacji wątków Pthreads
- Przetestowanie mechanizmu przesyłania argumentów do wątku
- Poznanie funkcjonowania obiektów określających atrybuty wątków

Zadanie:

1. Przygotowanie projektu
2. Uzupełnienie kodu programu *pthreads_detach_kill.c*

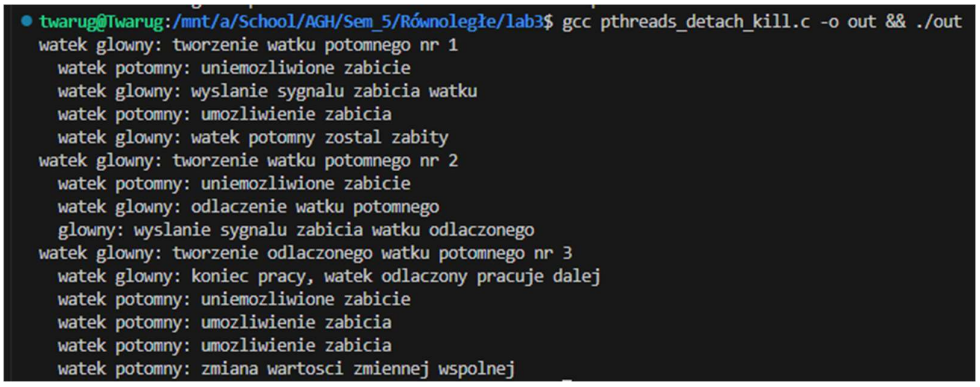
```
// inicjalizacja atrybutów
pthread_attr_init(&attr);
```

```
// tworzenie wątku potomnego z domyślnymi attr
pthread_create(&tid, NULL, zadanie_watku, NULL);
```

```
// odłączanie wątku
pthread_detach(tid);
```

```
// niszczenie atrybutów
pthread_attr_destroy(&attr);
```

```
// tworzenie odłączonego wątku
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, zadanie_watku, NULL);
```



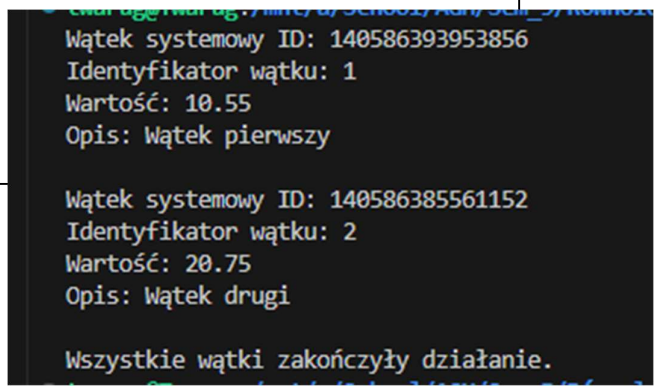
3. Implementacja nowego programu *pthreads_zadanie2.c*

```
void* print_id(void* arg) {
  sleep(1);
  int thread_id = *((int*)arg);
  printf("Wątek systemowy ID: %lu, Identyfikator przekazany: %d\n", pthread_self(), thread_id);
  return NULL;
}

int main() {
    pthread_t threads[N];
    int thread_ids[N];
    for (int i = 0; i < N; i++) {
        thread_ids[i] = i;
        if (pthread_create(&threads[i], NULL, print_id, &thread_ids[i]) != 0) {
            fprintf(stderr, "Błąd przy tworzeniu wątku %d\n", i);
            exit(1);
        }
    }
    for (int i = 0; i < N; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            fprintf(stderr, "Błąd przy oczekiwaniu na zakończenie wątku %d\n", i);
        }
    }
    printf("Wszystkie wątki zakończyły działanie.\n");
    return 0;
}
```

```
twarug@Twarug:/mnt/a/School/AGH/Sem_5/Równoległe/lab3$ gcc pthreads_zadanie2.c -o out && ./out
Wątek systemowy ID: 139992459880000, Identyfikator przekazany: 1
Wątek systemowy ID: 139992434701888, Identyfikator przekazany: 4
Wątek systemowy ID: 139992451487296, Identyfikator przekazany: 2
Wątek systemowy ID: 139992443094592, Identyfikator przekazany: 3
Wątek systemowy ID: 139992468272704, Identyfikator przekazany: 0
Wszystkie wątki zakończyły działanie.
```

4. Implementacja nowego programu *pthreads_zadanie3.c*

```
typedef struct {
   int threadId;
   float value;
   char description[50];
} ThreadData;

// Funkcja wykonywana przez wątek
void* print_structure(void* arg) {
   ThreadData* data = (ThreadData*) arg;
   printf("Wątek systemowy ID: %lu\n", pthread_self());
   printf("Identyfikator wątku: %d\n", data->threadId);
   printf("Wartość: %.2f\n", data->value);
   printf("Opis: %s\n\n", data->description);
   return NULL;
}

int main() {
   pthread_t threads[2];
   ThreadData threadData[2];

   threadData[0] = createThreadData(1, 10.55, "Wątek pierwszy");
   threadData[1] = createThreadData(2, 20.75, "Wątek drugi");

   // Tworzenie wątków
   for (int i = 0; i < 2; i++) {
      if (pthread_create(&threads[i], NULL, print_structure, &threadData[i]) != 0) {
         fprintf(stderr, "Błąd przy tworzeniu wątku %d\n", i + 1);
         exit(1);
      }
   }

   // Oczekiwanie na zakończenie wątków
   for (int i = 0; i < 2; i++) {
      if (pthread_join(threads[i], NULL) != 0) {
         fprintf(stderr, "Błąd przy oczekiwaniu na zakończenie wątku %d\n", i + 1);
      }
   }
   printf("Wszystkie wątki zakończyły działanie.\n");
   return 0;
}
```



Wnioski:

- Wątki odłączone automatycznie zwalniają zasoby po zakończeniu, co upraszcza zarządzanie pamięcią, podczas gdy wątki dołączone wymagają wywołania pthread_join() do ich odpowiedniego zakończenia.
- Wątki dołączone (joinable): Pozwalają na synchronizację z innymi wątkami za pomocą pthread_join(), co umożliwia kontrolowanie zakończenia i zwalnianie zasobów po zakończeniu. Są domyślnym typem wątków.
- Wątki odłączone (detached): Nie mogą być dołączane, a ich zasoby są automatycznie zwalniane po zakończeniu. Umożliwiają prostsze zarządzanie pamięcią, ale nie pozwalają na synchronizację z innymi wątkami.
- Przekazywanie danych do wątków powinno odbywać się za pomocą struktur lub dynamicznie alokowanej pamięci, aby uniknąć problemów z dostępem do usuniętej pamięci.

Odpowiedzi na pytania:

- Wątki Pthreads działają w trybach dołączonym (joinable) i odłączonym (detached). Dołączone wątki można kontrolować przez pthread_join(), a ich zasoby zwalniane są po zakończeniu. Kończą działanie, gdy osiągają koniec funkcji lub wywołują pthread_exit(). Można je zakończyć przez pthread_cancel(). Wątki odłączone nie mogą być dołączane i ich zasoby zwalniają się automatycznie. Wątki mogą bronić się przed zakończeniem, przechwytując sygnały lub zarządzając stanem, a pthread_testcancel() pozwala sprawdzić, czy zakończenie powiodło się.
- Aby poprawnie przesłać identyfikator do wątku w Pthreads, należy utworzyć strukturę z identyfikatorem i przekazać jej wskaźnik w pthread_create(). Użycie wskaźnika do liczby całkowitej może prowadzić do błędów synchronizacji, ponieważ zmienne lokalne mogą zostać usunięte po zakończeniu wątku, co skutkuje dostępem do nieprawidłowej pamięci. Zaleca się przesyłanie wskaźników do dynamicznie alokowanej pamięci lub zmiennych globalnych.
- W sprawozdaniu umieść dwa programy w C: pierwszy poprawnie przesyłający identyfikatory do wątków za pomocą struktury, a drugi z błędem synchronizacji, gdzie przekazywany jest wskaźnik do lokalnej zmiennej. Po uruchomieniu programów wykonaj zrzuty ekranu terminala z wynikami, z identyfikacją użytkownika terminala.