|  |  |  |
| --- | --- | --- |
| **Imię i Nazwisko**  Patryk Twardosz | **Kierunek**  Informatyka Techniczna | **Rok studiów i grupa**  I rok, Gr. 9 |
| **Data zajęć**  14.10.2024r. | **Numer i temat sprawozdania**  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();

1. Przeprowadzenie eksperymentu mierzącego czas tworzenie procesów i wątków.

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznieDebug Zoptymalizowane

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. 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);

}

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznieDebug Zoptymalizowane

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. 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*

// tworzenie odłączonego wątku

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

pthread\_create(&tid, &attr, zadanie\_watku, NULL);

// odłączanie wątku

pthread\_detach(tid);

// niszczenie atrybutów

pthread\_attr\_destroy(&attr);

// tworzenie wątku potomnego z domyślnymi attr

pthread\_create(&tid, NULL, zadanie\_watku, NULL);

// inicjalizacja atrybutów

pthread\_attr\_init(&attr);

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

1. 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;

}

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

   Opis wygenerowany automatycznieImplementacja 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.