Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium:

Pomiar czasu Clone / Fork

Przedmiot: Przetwarzanie Równoległe i Rozproszone

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: dr inż. Mateusz Sitko

Data: 10 października 2024

Numer lekcji: 2

Grupa laboratoryjna: 4

Cel Ćwiczenia

Celem ćwiczenia było dokonanie eksperymentów, pozwalających na lepsze zapoznanie się z cechami charakterystycznymi funkcji **clone** oraz **fork** służących do tworzenia wątków i procesów, w tym przeprowadzenie pomiarów czasowych ich tworzenia.

Przebieg Ćwiczenia

Ze strony internetowej przedmiotu zostały pobrane między innymi dwa proste pliki źródłowe, **fork.c** oraz **clone.c** napisane w języku **C**, realizujące tworzenie 1000 procesów oraz 1000 wątków w systemie **Linux**. Powyższe kody źródłowe zostały wzbogacone o procedury pomiaru czasu. Dodatkowo, liczba tworzonych procesów/wątków została zwiększona dziesięciokrotnie (10 000), aby otrzymane pomiary czasowe na maszynach o większej mocy obliczeniowej nie zbiegały się do wartości zerowych.

```
inicjuj_czas();
double cpu_start = czas_CPU();
double clock_start = czas_zegara();

for(i=0;i<10000;i++)
{

  pid = fork();

  if(pid==0)
  {
    zmienna_globalna++;
    char* arg[] = {"./program2", "Spongebob", "Squarepants", NULL};
    wynik=execv(arg[0], arg);
    if(wynik==-1)
      printf("Proces potomny nie wykonal programu\n");

    exit(0);

  }
  else
  {

    wait(NULL);

  }

}

//drukuj_czas();
double cpu_end = czas_CPU();
double clock_end = czas_zegara();
```

Fragment pliku fork.c

```
// Poczatek mierzenia czasu
inicjuj_czas();
double cpu_start = czas_CPU();
double clock_start = czas_zegara();

for(i=0;i<10000;i++){

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

  waitpid(pid, NULL, __WCLONE);

}

// Koniec mierzenia czasu
double cpu_time = czas_CPU() - cpu_start;
double clock_time = czas_zegara() - clock_start;
```

Fragment pliku clone.c

Pomiary zostały przeprowadzone kilkukrotnie na dwóch różnych, skompilowanych przez **GCC,** wersjach kodu źródłowego: z zerową optymalizacją i opcjami debugowania (**-g -DDEBUG -O0**) oraz maksymalną optymalizacją (**O3**).

W poniższej tabeli przedstawione zostały uśrednione wyniki:

Próby	Funkcja	Kompilacja	Średni czas zegara pojedynczego wywołania	Średni czas CPU pojedynczego wywołania
4	Clone	-g -DDEBUG -O0	3.23024E-05	6.042E-07
4	Clone	-O3	3.15786E-05	4.603E-07
4	Fork	-g -DDEBUG -O0	1.86797E-4	2.85703E-06
4	Fork	-O3	1.96893E-4	1.99993E-06

Tabela 1

Na podstawie uzyskanych wyników możemy zauważyć, że optymalizacja miała znaczący wpływ na czas CPU. Czas zegara jednak uległ wyłącznie minimalnym zmianom, w przypadku **fork** nawet nieznacznie rosnąc.

Otrzymane wyniki są rezultatem tego, że optymalizacja kompilatora polega na przyspieszeniu czasu jaki procesorowi zajmie wykonywanie instrukcji. Wpływ optymalizacji na czas zegarowy jest minimalny, ponieważ jest on również zależny od innych czynników systemowych, takich jak współbieżność procesów lub inne operacje systemowe. Wyniki w *Tabeli 1* zostały porównanie z wynikami pomiaru czasowego z poprzedniej lekcji.

W poniższych tabelkach zostało zawarte ile wybranych działań mógł dokonać system w trakcie wywołania Clone lub Fork. Obliczenia zostały dokonane na podstawie czasu zegara.

W trakcie wywołania Clone (O3)	
Operacje Arytmetyczne	127.61
Operacje Wejścia/Wyjścia	13049
Wywołania Fork	0.16

Tabela 2

W trakcie wywołania Fork (O3)	
Operacje Arytmetyczne	795.68
Operacje Wejścia/Wyjścia	81360.82
Wywołania Clone	6.23

Tabela 3

Dodatkowo przeprowadzony został jeszcze eksperyment mierzący czas realizacji funkcji **execv** wewnątrz utworzonego wątku. Przed wywołaniem funkcji zapisano czas zegara oraz CPU i podano go jako argument wywoływanemu przez funkcje programowi, który obliczył różnice i wypisał wyniki.

```
int funkcja_watku( void* argument )
{

  zmienna_globalna++;

  // Przygotowanie argumentów do execv
  char clock_time[32];
  char cpu_time[32];

  // Zapisz czas
  struct timespec clock, CPU;
  clock_gettime(CLOCK_REALTIME, &clock);
  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &CPU);

  // Konwersja czasu na string
  snprintf(clock_time, sizeof(clock_time), "%ld.%09ld", clock.tv_sec,
clock.tv_nsec);
  snprintf(cpu_time, sizeof(cpu_time), "%ld.%09ld", CPU.tv_sec,
CPU.tv_nsec);

  char* args[] = {"./program", clock_time, cpu_time, NULL};

  // Wywołanie execv
  int wynik = execv(args[0], args);
  if(wynik==-1)
    printf("Proces potomny nie wykonal programu\n");

  return 0;
}
```

Funkcja wywoływana przez wątek

```c
int main(int argc, char *argv[])
{
    // Save current time
    struct timespec clock_end, cpu_end;
    clock_gettime(CLOCK_REALTIME, &clock_end);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_end);

    double clock_time_end = clock_end.tv_sec + clock_end.tv_nsec /
1000000000.0;
    double CPU_time_end = cpu_end.tv_sec + cpu_end.tv_nsec / 1000000000.0;

    // Check for correct argument list
    if (argc != 3)
    {
        printf("Niepoprawna liczba argumentow\n");
        return -1;
    }

    // Read time given as arguments
    double clock_time_start = atof(argv[1]);
    double cpu_time_start = atof(argv[2]);

    // Get difference
    double elapsed_clock = clock_time_end - clock_time_start;
    double elapsed_cpu = cpu_time_start - cpu_time_start;

    // Print results
    printf("---------------------------------\n");
    printf("Czas Zegara:\t%.14lf\n", elapsed_clock);
    printf("Czas CPU:\t%.14lf\n", elapsed_cpu);

}
```

*Funkcja **main** programu uruchamianego przez **execv***

W powyższych kodach źródłowych bardzo istotne jest zwrócenie uwagi na to kiedy zaczyna i kończy się pomiar czasu.

Uzyskane średnie wyniki z 10 prób są następujące:

- Czas CPU: 0
- Czas zegara: 5.17E-04

Wnioskiem z przeprowadzonego testu jest to, że czas, jaki procesor potrzebuje na uruchomienie funkcji **execv**, jest praktycznie pomijalny, o czym świadczy zerowy czas CPU. Natomiast zauważalnie dłuższy czas zegara w porównaniu do funkcji **fork** i **clone** sugeruje, że pewien moment upływa na przekazaniu programu do systemu operacyjnego i jego przygotowanie do wykonania, zanim procesor faktycznie rozpocznie pracę. To pokazuje, że pomimo, iż sam procesor działa bardzo szybko, inne operacje systemowe, takie jak zarządzanie procesami i wątkami, mogą mieć zauważalny wpływ na całkowity czas wykonania.

Kolejnym zadaniem było napisanie programu, w którym bezpośrednio po sobie tworzone są dwa wątki do działania równoległego. W pętli wykonującej 100 000 iteracji, wątki miały zwiększać wartości zmiennej lokalnej, przekazanej jako argument oraz globalnej. Oba wątki po wyjściu z pętli wypisały wartości obu zmiennych, ich wydruki zostały porównane dodatkowo z finalnym wynikiem z funkcji **main** programu.

```c
int main()
{
    // Memory allocation
    void* stack1 = malloc(STACK_SIZE);
    void* stack2 = malloc(STACK_SIZE);
    if (stack1 == NULL || stack2 == NULL)
    {
        printf("Issues with stack allocation\n");
        return -1;
    }

    // Thread arguments
    int arg1 = 0;

    // Creating threads
    pid_t thread1, thread2;
    thread1 = clone(thread_func, stack1 + STACK_SIZE, CLONE_VM, &arg1);
    thread2 = clone(thread_func, stack2 + STACK_SIZE, CLONE_VM, &arg1);

    // Waiting for thread execution
    waitpid(thread1, NULL, __WCLONE);
    waitpid(thread2, NULL, __WCLONE);

    // Outputting results
    printf("----------MAIN----------\n");
    printf("global_var: %d\n", global_var);
    printf("arg: %d\n", arg1);

    // Memory cleanup
    free(stack1);
    free(stack2);

    return 0;
}
```

*Funkcja **main** kodu **threads.c***

```c
int thread_func(void* arguments)
{
    int local_var = *((int*)arguments);

    for (int i = 0; i < 100000; i++)
    {
        global_var++;
        local_var++;
    }

    printf("----------THREAD----------\n");
    printf("global_var: %d\n", global_var);
    printf("local_var: %d\n", local_var);

    return 0;
}
```

*Funkcja przekazywana wywołanym wątkom w **threads.c***

Test został przeprowadzony na dwóch różnych wersjach programu. Oryginalna wersja niepoprawnie zajmowała się zmienną lokalną.

```
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ./threads
----------THREAD----------
----------THREAD----------
----------THREAD----------
global_var: 200000
global_var: 200000
global_var: 200000
local_var: 100000
local_var: 100000
----------MAIN----------
global_var: 200000
arg: 0
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ./threads
----------THREAD----------
global_var: 100000
local_var: 100000
----------THREAD----------
global_var: 200000
local_var: 100000
----------MAIN----------
global_var: 200000
arg: 0
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ./threads
----------THREAD----------
global_var: 100000
local_var: 100000
----------THREAD----------
global_var: 200000
local_var: 100000
----------MAIN----------
global_var: 200000
arg: 0
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$
```

*Wydruk nowej wersji **threads.c***

```
----------THREAD_NUM: 1----------
global_var: 119468
local_var: 120072
----------THREAD_NUM: 1----------
global_var: 176432
local_var: 176230
----------MAIN----------
global_var: 176432
args1->local_var: 176230
args2->local_var: 0
```

*Wydruk starej wersji **threads.c***

Powodem dlaczego chciałbym przedstawić wydruk starszej wersji programu jest to, iż nie udało mi się odtworzyć tzw. race condition, na zmiennej globalnej, w nowszej wersji programu, nawet przy drastycznym zwiększeniu ilości iteracji i obciążenia maszyny.

Obserwacje:

Wydruk:

Możemy zauważyć dosyć spore problemy z wydrukiem, gdzie pewne ciągi znaków są wypisywane w nieodpowiedniej kolejności i w nieodpowiednim momencie co dzieje się dlatego, że brakuje synchronizacji pomiędzy zasobami wejścia/wyjścia i wątki „ścigają" się o dostęp do strumienia wyjścia co prowadzi do nieoczekiwanych wyników.

Zmienne lokalna

Zmienne lokalne, które są kopiami zmiennej z funkcji **main**, przekazanymi wątkom przez wskaźnik, mają poprawne wartości. Ponieważ żaden z wątków nie modyfikuje oryginalnej pamięci otrzymany wynik jest zgodny z oczekiwaniami.

Zmienna Globalna

Zmienna globalna zwiększa swoją wartość o 100 000 w każdym wątku i kończy z poprawną wartością w funkcji **main**. Pomimo, iż wynik ten jest poprawny, nie jest on zgodny z oczekiwanym scenariuszem. W oczekiwanym rezultacie, jak można było zobaczyć w starszej wersji programu, wyniki były „bezsensowne", co wynika z problemu tzw. wyścigu wątków (race condition). Dzieje się tak, ponieważ oba wątki próbują modyfikować globalną zmienną w tym samym czasie.

W tej sytuacji każdy wątek może „ścigać się" o to, który z nich nadpisze wartość zmiennej jako pierwszy. Można sobie wyobrazić sytuację, w której jeden z wątków odczytuje bieżącą wartość zmiennej globalnej, zwiększa ją o jeden, ale zanim zdąży nadpisać wartość, drugi wątek również odczytuje ją przed modyfikacją. W efekcie oba wątki pracują na przestarzałych danych, co prowadzi do utraty części inkrementacji.

W takiej sytuacji można by oczekiwać, że globalna zmienna osiągnie wartość 100 000 (czyli wynik jednej iteracji), ponieważ jeden wątek mógłby wielokrotnie nadpisywać zmiany dokonane przez drugi. Jednak nawet to nie jest prawdą, ponieważ najmniejsze różnice w prędkości wykonywania wątków, wynikające z warunków systemowych, sprawią, że wynik będzie zupełnie nieprzewidywalny.

W przypadku nowej wersji programu, race condition może nie występować z powodu specyfiki harmonogramu systemowego, który przydziela wątkom czas procesora w taki sposób, że ich operacje na zmiennej globalnej są wykonywane sekwencyjnie, a nie równolegle. Pomimo korzystania z tej samej maszyny, nawet starsza wersja programu skompilowana ponownie nie przyniosła oczekiwanych (tj. niezsynchronizowanych) rezultatów.

Po dalszym badaniu powodu otrzymywanych wyników, okazało się, że za pośrednictwem programu **make** kompilator pracował z ustawieniem optymalizacji -O3, które powodowało poprawność wyników. Po zmniejszeniu poziomu optymalizacji otrzymano oczekiwany rezultat.

```
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ./threads
----------THREAD----------
global_var: 13171281
local_var: 10000000
----------THREAD----------
global_var: 15029352
local_var: 10000000
----------MAIN----------
global_var: 15029352
arg: 0
```

Wydruk z nowej wersji programu bez optymalizacji -O3

Agresywna optymalizacja zastosowana przez kompilator prawdopodobnie zastąpiła iteracyjną pętlę pojedynczą operacją arytmetyczną. Ponieważ operacje arytmetyczne są znacznie szybsze niż tworzenie wątku (tabela 2), bardzo możliwe jest, że pierwszy utworzony wątek zakończył swoje obliczenia, zanim drugi wątek rozpoczął pracę. Natomiast operacje wyjścia funkcji **printf**, pomimo że są z natury szybsze niż operacje arytmetyczne, napotkały problemy z race condition, prawdopodobnie spowodowane konfliktem wątków o dostęp do strumienia wyjścia.

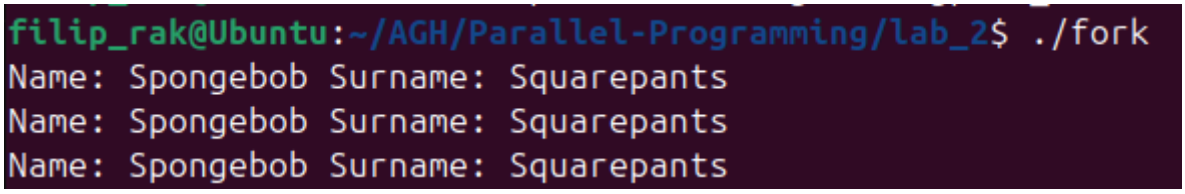
W ramach dalszych zadań zmodyfikowano kod źródłowy **fork.c**, w taki sposób aby wywoływany przez wątek, za pośrednictwem funkcji **execv**, program mógł otrzymać argumenty, które następnie wydrukuje w terminalu.

```c
if(pid==0)
{
	zmienna_globalna++;
	char* arg[] = {"./program2", "Spongebob", "Squarepants", NULL};
	wynik=execv(arg[0], arg);
	if(wynik==-1)
		printf("Proces potomny nie wykonal programu\n");

	exit(0);

}
```

*Fragment kodu źródłowego **fork.c***

```c
int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        printf("PROGRAM2::MAIN -> Argument number is off\n");
        return -1;
    }

    printf("Name: %s\tSurname: %s\n", argv[1], argv[2]);
    return 0;
}
```

*Fragment kodu źródłowego wywoływanego programu, **program2.c***



Fragment uzyskanego wydruku

Dalszym etapem zajęć było przeanalizowanie funkcji **clone** i modyfikacja przekazywanych do niej argumentów w sposób taki aby jej działanie pokrywało się z funkcją **fork**.

Przykładowe wywołanie funkcja **clone** może wyglądać w następujący sposób:

```
pid = clone(funkcja, stos, flagi, argumenty);
```

Gdzie:

- `pid` jest identyfikatorem utworzonego wątku, lub -1 w przypadku niepowodzenia.
- `funkcja` jest wskaźnikiem na funkcję, którą będzie wykonywana przez wątek.
- `stos` to adres końca stosu (adres stosu + jego rozmiar), który będzie używany do przechowywania zmiennych lokalnych wątku.
- `flagi` to parametry sterujące zachowaniem nowego wątku, przykładowo:
  - **CLONE_VM** - współdzielona przestrzeń adresowa.
  - **CLONE_FS** - współdzielenie systemu plików.
  - **CLONE_FILES** - współdzielenie otwartych plików.
  - **CLONE_SIGHAND** - współdzielenie uchwytów sygnałów.
- `argumenty` to parametry przekazane do uruchamianej przez wątek funkcji.

W celu wywołania funkcji **clone** w taki sposób, aby jej działanie pokrywało się z funkcją **fork**, tworzącą proces, konieczne jest pozbycie się flag pozwalających na współdzielenie zasobów oraz zastosowanie flagi **SIGCHLD**, która sprawia, że proces potomny będzie zgłaszał sygnał zakończenia. Dzięki temu nowo utworzony proces będzie miał swoją własną, niezależną przestrzeń adresową, pliki i uchwyty sygnałów, podobnie jak w przypadku fork.

Poniżej znajduje się fragment kodu realizujący wcześniejszy opis:

```
pid_t pid = clone(thread_func4, stack + STACK_SIZE, SIGCHLD, NULL);

if (pid == -1)
{
	perror("CLONE_TEST.C::EX4 -> Watek nie utworzony\n");
	return;
}

waitpid(pid, NULL, 0);
```

Fragment kodu źródłowego clone_test.c

Przeprowadzona została dalsza analiza stosu, w świetle dwóch kolejnych eskperymentów. Podczas pierwszego z nich zadeklarowana została tablica statyczna w funkcji wywoływanej przez wątek.

```
int thread_func5_1(void* arg)
{
    const int arr_size = 1900;  // 1900 to limit dla 8192 // 8192 / 1900 = 4.31157894737
    //const int arr_size = 100;  // Wykorzystanie printf obniza limit
    //const int arr_size = STACK_SIZE / 4.18279295379 -1; // 4.18279295379 -1
    int arr[arr_size];
    for (int i = 0; i < arr_size; i++)
        arr[i] = i;


    printf ("CLONE_TEST.C::THREAD_FUNC5_1 -> Watek wykonal prace\n");
}
```

Fragment kodu clone_test.c

Przeprowadzony próby pokazały, że wymagany rozmiar stosu jest w pewnym stopniu dosyć trudny do przewidzenia. Oczywistym wydawało by się to, że jeżeli nasz stos ma rozmiar **x** bitów, to rozmiar tablicy 4-bitowych zmiennych typu **int** powinien wynosić **x** / 4, jednak w praktyce okazało się, że wątek wymaga jeszcze dodatkowej pamięci, która nie jest tak prosta do wyliczenia. Drobne przekroczenie pamięci owocowało w błędach w terminalu, drastyczne przekroczenie pamięci jednak nie powodowało żadnego wydruku.

Drugi przeprowadzony test polegał na uruchomieniu funkcji iteracyjnej, bez warunku stopu, która alokowała 4 bity pamięci i wypisywała poziom rekurencji do konsoli.

```
void recursive_function(int count)
{
    int temp[1];

    printf("Poziom rekurencji: %d\n", count);

    recursive_function(count + 1);
}
```

Fragment clone_test.c

W efekcie program wypisywał w konsoli poziom rekurencji tak długo jak był w stanie pracować a następnie zawieszał się. Uzyskane wyniki pokazały, że zmiana rozmiaru stosu prowadziła do proporcjonalnego zwiększenia ilości rekurencji z pewnymi odchyleniami, które wynikały z wymogu dodatkowych zasobów systemowych wykorzystywanych przez wątek, takich jak zarządzanie stosami, mechanizmy obsługi sygnałów oraz inne operacje systemowe, które zajmują część pamięci przeznaczonej na stos.

```
CLONE_TEST.C::RECURSIVE_FUNCTION -> Poziom rekurencji: 143
CLONE_TEST.C::RECURSIVE_FUNCTION -> Poziom rekurencji: 144
```

Ostatnie linijki wydruku funkcji rekurencyjnej z testu dla stosu o rozmiarze 8192 bitów

```
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ulimit -s
8192
```

Domyślny rozmiar stosu na użytej maszynie

Ostatnim zadaniem była próba otwarcia pliku w przypadku braku współdzielenia systemu plików (flaga **CLONE_FS**).

Przygotowany został prosty test. Utworzony został plik testowy w katalogu innym niż program. Zadaniem utworzonego w programie wątku było, z wykorzystaniem funkcji **chdir**, zmiana bieżącego katalogu roboczego na katalog zawierający plik testowy. Proces nadrzędny następnie po zakończeniu pracy wątku podejmował próbę otworzenia pliku, po jego nazwie.

```
pid_t pid = clone(thread_func5_3, stack + STACK_SIZE, SIGCHLD, NULL);

if (pid == -1)
{
    perror("CLONE_TEST.C::EX5_3 -> Watek nie utworzony\n");
    return;
}

waitpid(pid, NULL, 0);

// Proba otwarcia pliku w nowym katalogu
FILE* fd = fopen("testfile", "r");
if (!fd)
{
    perror("CLONE_TEST.C::EX5_3 ->");
    return;
}

printf("CLONE_TEST.C::EX5_3 -> Plik zostal otworzony\n");
fclose(fd);
```

Fragment kodu wykonywanego przez proces nadrzędny

```c
int thread_func5_3(void* arg)
{
    // Zmiana katalogu
    if (chdir("test_dir") != 0)
    {
        perror("CLONE_TEST.C::THREAD_FUNC5_3 -> chdir failed\n");
        return -1;
    }

    return 0;
}
```

Fragment kodu wykonywanego przez utworzony wątek

```
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ./clone_test
CLONE_TEST.C::EX5_3 ->: No such file or directory
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ make
gcc -g -DDEBUG -O0 clone_test.c -o clone_test
filip_rak@Ubuntu:~/AGH/Parallel-Programming/lab_2$ ./clone_test
CLONE_TEST.C::EX5_3 -> Plik zostal otworzony
```

*Wydruk przed i po dodaniu flagi **CLONE_FS***

Przeprowadzony test potwierdza wpływ flagi **CLONE_FS** na synchronizacje systemu plików pomiędzy wątkami. Wątek nadrzędny zsynchronizował się ze zmianą katalogu roboczego wprowadzoną przez wątek podrzędny, dzięki czemu otworzenie pliku powiodło się. Brak flagi **CLONE_FS** sprawił, że katalog roboczy nie został zmieniony a próba otwarcia pliku nie powiodła się.