Grupa lab.
5

Data ćwiczenia
9.10.2023

Laboratorium 2

Imie i nazwisko
Łukasz Sawina

Pomiar czasu wykonania 1000 procesów/1000 wątków

Wyniki pomiarów czasów programu *fork.c*

FORK()	DEBUG		OPTYM	
Lp.	Zegar [s]	CPU [s]	Zegar [s]	CPU [s]
1	0.597	0.0121	0.661	0.00906
2	0.664	0.0037	0.556	0.00307
3	0.542	0.0154	0.404	0.00181
4	0.629	0.0054	0.607	0.01528
5	0.652	0.0135	0.607	0.00977
Średnia	**0.617**	**0.0137**	**0.567**	**0.00593**

Wyniki pomiarów czasów programu *clone.c*

CLONE()	DEBUG		OPTYM	
Lp.	Zegar [s]	CPU [s]	Zegar [s]	CPU [s]
1	0.0643	0.00120	0.0351	0.00699
2	0.0781	0.00359	0.0801	0.00194
3	0.0722	0.00310	0.0727	0.00273
4	0.0708	0.01969	0.0562	0.00469
5	0.0880	0.00686	0.0713	0.01516
Średnia	**0.0747**	**0.00368**	**0.0701**	**0.00409**

Wartości zaznaczone kolorem czerwonym zostały odrzucone, z powodu dużych różnic w porównaniu do pozostałych wartości

Zestawienie uśrednionych wartości pomiarów

	DEBUG		OPTYM	
	Zegar [s]	CPU [s]	Zegar [s]	CPU [s]
FORK()	**0.617**	**0.0137**	**0.567**	**0.00593**
CLONE()	**0.0747**	**0.00368**	**0.0701**	**0.00409**
ARYTMETYCZNE	**0.000368**	**0.000355**	**0.000393**	**0.000303**
WEJŚCIA/WYJŚCIA	**0.0414**	**0.0119**	**0.0481**	**0.0173**

Jak widać na zestawieniu różnica między działaniem *fork()* oraz *clone()* jest znaczna, jest to spowodowane różnicą w samym działaniu procesów oraz wątków. Narzut na tworzenie wątków jest około 0.121 razy mniejszy niż narzut na tworzenie procesów. Jest to spowodowane tym, że wątki są mniej wymagające w zasobach systemowych niż procesy. Tworzenie procesu wymaga nowego środowiska procesowego, co jest kosztowne, dodatkowo wątki współdzielą przestrzeń adresową oraz zasoby. Dodatkowo jak widać optymalizacja w przypadku czasu zegara przyspieszyła samo działanie, jednak dla CPU z pomiarów wynika, że lekko spowolniła.
W czasie tworzenia jednego wątku możliwe jest wykonanie ok. 203 operacji arytmetycznych oraz ok. 2 operacji wejścia wyjścia w trybie debugowania, a w zoptymalizowanej 178 operacji arytmetycznych oraz ok. 1 operację wejścia/wyjścia.

Utworzenie dwóch wątków działających w sposób równoległy

Do utworzenia dwóch wątków działających równolegle potrzebne było tak naprawdę zrobienie dodatkowej kopii wszystkiego co znajdowało się w funkcji *main* w pliku *clone.c*, w taki sposób mogliśmy utworzyć dwa wątku działające równolegle.

```
 pid1 = clone( &funkcja_watku, (void *) stos1+ROZMIAR_STOSU,
                    CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, &parametr1 );
 pid2 = clone( &funkcja_watku, (void *) stos2+ROZMIAR_STOSU,
                    CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, &parametr2 );
 waitpid(pid1, NULL, __WCLONE);
 waitpid(pid2, NULL, __WCLONE);
 printf("Wartość zmiennej przekazanej do wątku 1 na koniec pętli: %d\n", parametr1);
 printf("Wartość zmiennej przekazanej do wątku 2 na koniec pętli: %d\n", parametr2);
 printf("Wartość zmiennej globalnej na koniec pętli: %d\n",zmienna_globalna);
```

Ważne, aby funkcje *waitpid()* były wykonane na końcu po utworzeniu wątków. Oba procesy otrzymują swoje własne stosy oraz na końcu przekazane są parametry, oddzielne dla każdego, aby lepiej było zobaczyć działanie wątków. Oczywiście na końcu programu pamięć ze stosu jest uwalniana.

Oba procesy wykonują tą samą funkcję funkcja_watku(), do której przekazywana jest wartość początkowa dla zmienna_przekazana, która jest pobierana z parametrów funkcji i rzutowana na int.

```
int funkcja_watku( void* argument )
{
 int i = 0;
 int zmienna_porzekazana = *(int*)argument;
 for(i; i < 1000; i++)
 {
  zmienna_globalna++;
  zmienna_porzekazana++;
 }
   printf("Wartość zmiennej przekazanej do wątku %d na koniec pętli: %d\n", (zmienna_porzekazana-1000)/10,
zmienna_porzekazana);
 printf("Wartość zmiennej globalnej na koniec pętli: %d\n\n",zmienna_globalna);
 return 0;
}
```

Wyniki programu:

```
lukasz@lukasz-ThinkPad:~/C_programs/Lab_2$ ./clone_copy
Wartość zmiennej przekazanej do wątku 2 na koniec pętli: 1020
Wartość zmiennej przekazanej do wątku 1 na koniec pętli: 1010
Wartość zmiennej globalnej na koniec pętli: 2000

Wartość zmiennej globalnej na koniec pętli: 2000

Wartość zmiennej przekazanej do wątku 1 na koniec pętli: 10
Wartość zmiennej przekazanej do wątku 2 na koniec pętli: 20
Wartość zmiennej globalnej na koniec pętli: 2000
```

Jak widać na koniec programu wartość zmiennej globanej wynosi 2000, co pasuje do działania naszej funkcji, iteruje ona 1000 razy i zwiększa wartość każdej zmiennej o 1. Dodatkowo widać, że zmienne przekazane do funkcji zwiększyły swoją wartość na koniec pętli, jednak w funkcji main dalej pozostały w pierwotnej wersji. Zmienna globalna oraz lokalna zachowuje się tak jak przy wywołaniu zwykłej funkcji, jednak zostały one uruchomione równolegle.

Wartość (zmienna_porzekazana-**1000**)/**10** służy do wypisania na konsoli numeru wątku który został wywołany, zmienna przekazana do funkcji jest odpowiednio 10 i 20 dla pierwszego i drugiego wątku.

Uruchomienie programu ./program za pomocą funkcji exec

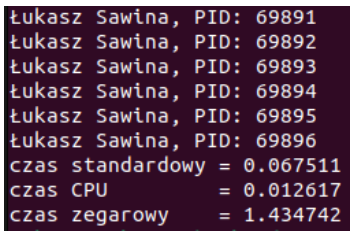
Utworzyłem program wyświetlający imię, nazwisko oraz numer przekazany jako argument przy wywołaniu programu:

```
int main(int argc,char *argv[])
{
   int pid = atoi(argv[1]);
   printf("Łukasz Sawina, PID: %d\n", pid);
   return 0;
}
```

W programie *clone.c* zmieniłem funkcję funkcja_watku, aby uruchamiała ./program z przekazanymi parametrami

```
int funkcja_watku( void* argument )
{
  zmienna_globalna++;
  int wynik;
  char pid_nr[20];
  sprintf(pid_nr, "%d", getpid());
  char *arguments[] = {"./program", (char *)pid_nr, NULL};
  wynik=execv("./program", arguments);
  if(wynik==-1)
    printf("Proces potomny nie wykonal programu\n");
  return 0;
}
```

Funkcja pobiera numer pid wątku przy pomocy *getpid()* i konwertuje go na tablicę char przy wykorzystaniu *sprint()* oraz przesłane w wywołaniu execv. Ważne aby nasza tablica argumentów była zakończona NULL.

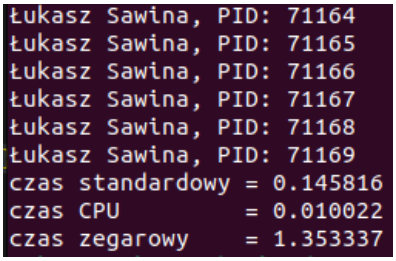


Oczywiście to jest tylko urywek, funkcja wykonywana jest 1000 razy. Jak widać udało sie uruchomić inny program w osobnym wątku oraz przekazać do niego parametry.

W programie fork.*c* zmiany są bardzo podobne, tylko nie w funkcji

```
char pid_nr[20];
sprintf(pid_nr, "%d", getpid());
char *arguments[] = {"./program", (char *)pid_nr, NULL};
wynik=execv("./program", arguments);
if(wynik==-1)
   printf("Proces potomny nie wykonal programu\n");
```

Działanie programu jest identyczne jak *clone.c*, wynik jest również podobny.



Wnioski

Ćwiczenie w całkiem dobry sposób pokazało działanie tworzenia wątków i procesów oraz różnicę miedzy nimi. Przede wszystkim różnicę w czasie wykonywania oraz potrzebnych zasobów. W dobry sposób również pokazało jak wykonywać pewne operacje równolegle z wykorzystaniem *clone()* oraz możliwość wywoływania innych programów wewnątrz naszego programu w osobnym procesie/wątku. Ćwiczenie różnież pokazało jak przy pomocy *exec()* wywołać inny program wraz z przekazanymi do niego parametrami, co na pewno jest przydatne, jeśli chcemy aby nasze programy miały bardziej złożone opcje działania.