Make | Pomiar czasu | clone() / fork()
Andrzej Janaszek (ITE)
16.10.2025

MAKE I KOMPILACJA

Make automatyzuje kompilację programu składającego się z kilku plików źródłowych. Automatycznie śledzi zależności dla danego targetu i wywołuje potrzebne zależności (targety). Po zmianie pliku wszystkie od niego zależne zostaną skompilowane od nowa.

Struktura Makefile:
[*nazwa celu*]: [*zależności*]
<tab> [*komenda*]
przykład:
```
main: main.c
     gcc main.c -o main
```

uruchomienie w terminalu: `make main`

Kompilacja pliku (kodu .c -> maszynowy .o)
-c : kompiluj
-o : do pliku o nazwie
gcc -c plik.cpp o- plik.o

Pliki nagłówkowe / includes:
-I (duże i)
biblioteki:
-Lfolder -lplik1 -lplik2 (L potem l czyli małe L)

Zmienne:
```
zmienna = wartość
```
wykorzystanie:
```
$(zmienna)
```

POMIAR CZASU

Sposoby mierzenia czasu.

Ogólnie: a) czas CPU, b) czas rzeczywisty.

Proponowane metody mierzenia czasu:
1. clock() - takty procesora (ticks)
2. getrusage() - systemowe informacje o wykonywanym procesie (tutaj ru_utime czyli czas użytkownika, ewentualnie ru_stime czas systemowy)
3. gettimeofday() - czas zegarowy / czas rzeczywisty (wall clock time)

clock() [zdefiniowana w <time.h>]
Zwraca przybliżoną ilość taktów procesora (clock ticks) od początku startu procesu. Aby otrzymać znaczący czas musimy zrobić dwa znaczniki czasu (timestamp) początkowy i końcowy, następnie policzyć różnice między nimi, ponieważ początek zliczania taktów niekoniecznie musi być równy startowi programu. Tzw. era (początek liczenia taktów) jest zależny od implementacji.
Aby otrzymać wartość w sekundach należy różnice znaczników czasu podzielić przez makro CLOCKS_PER_SEC.[2]

W zależności od użycia procesora czas może różnić się względem czasu rzeczywistego.
Np. jeżeli program będzie wykonywany jednowątkowo przy obciążeniu to czas rzeczywisty będzie dłuższy. Jeżeli jednak program zostanie rozbity na kilka wątków przy małym obciążeniu procesora czas rzeczywisty może być mniejszy ponieważ czas cpu to suma czasu dla każdego wątku. [1],[2]

```
#include<time.h>
// ...
clock_t start = clock();
// program
clock_t end = clock();

double diff = (double)(end - start) / (double)CLOCKS_PER_SEC
```

getrusage() [zdefiniowana w <sys/resource.h>]
Funkcja ta zwraca informacje od systemu na temat procesu. Interesujące nas parametry to ru_utime (czas użytkownika, user time) oraz ewentualnie ru_stime (czas systemu, system time).
RUSAGE_SELF - zwraca informacje o aktualnym procesie
RUSAGE_CHILDREN - zwraca informacje o procesach potomnych (zakończonych i w czasie wykonywania) [3],[4]

Czas użytkownika to czas procesu w trybie użytkownika inaczej mówiąc kod programu (instrukcje warunkowe, pętle itd.), a czas systemu to czas spędzony na obsłudze przez kernel np. dostęp do plików, alokacja pamięci itd.
Czasy te są w strukturze timeval (sekundy plus mikrosekundy).[3]
Są to również czasy procesora (CPU time) tak jak w funkcji clock().

gettimeofday() [#include <sys/time.h>]
W skrócie pobiera czas zegarowy systemu. Oznacza to że mierząc w taki sposób czas wykonania programu musimy brać pod uwagę, że mierzymy wszelkie inne takie jak przełączanie się procesów w systemie.
Prosty przykład

pseudokod:

```
start czas zegarowy
for i in range(0, 1000000)
     inkrementuj zmienna
sleep(60)
koniec czasu zegarowego
```

Taki pomiar zwróci nam: czas wykonania pętli + 60s, podczas gdy czas CPU zwróciłby w przybliżeniu jedynie czas wykonania pętli.

Jak podejść do mierzenia czasu? Problemy oraz co może wpłynąć na czas wykonywania.

Na otrzymane wyniki pomiarów mogą wpłynąć różne czynniki np.:

- liczba operacji których czas mierzymy (jeżeli jedna operacja trwa kilka cykli zegara to bardzo trudno będzie to zmierzyć. Musimy zatem zmierzyć czas N takich operacji i tak otrzymany czas podzielić przez N aby otrzymać wartość dla jednego wykonania)
- inne działania systemu w tle (system inne programy)
- obciążenie procesora
- optymalizacja kompilatora
- czas operacji IO i arytmetycznych
- wielowątkowość

Aby zminimalizować niepewność pomiarową musimy wykonać odpowiednią liczbę pomiarów oraz usunąć skrajne wartości pomiarów (pewnie dobrze byłoby również zastanowić się skąd się one biorą).
Trzeba być świadomym jak optymalizacja kompilatora (flaga -O) wpływa na generowany wykonywalny. Optymalizacja może “skrócić” pętle dodającą wartość do pewnej zmiennej i zastąpić to po prostu przypisaniem ostatecznego wyniku do danej zmiennej. Nie zmierzymy wtedy czasu wykonania pętli a jedynie pojedynczy czas przypisania.

Wpływ obciążenia procesora możemy raczej w dość dobrym stopniu wykluczyć przez ilość pomiarów wyciągnięcie z nich średniej.
Rozłożenie programu na wątki również może mieć wpływ na dokonane pomiary co opisane zostało wyżej przy okazji funkcji clock().

Pomiary na przykładzie operacji IO oraz arytmetycznych

Pomiaru dokonujemy najpierw dla pętli z operacjami IO, następnie dla pętli z operacjami arytmetycznymi.

Przykłady pomiarów na podstawie tworzenia wątków i procesów (clone i fork)
Pomiary dla tworzenia procesów funkcją fork() w trybie debug oraz wyciągnięcie z nich średniej.

DEBUG FORK			
l.p.	czas standardowy	czas CPU	czas zegarowy
1	0.045395	0.000000	0.119501
2	0.045178	0.000736	0.119098
3	0.045963	0.000659	0.118947
4	0.046063	0.000000	0.118861
5	0.046061	0.000951	0.120271
6	0.046204	0.001746	0.121987
7	0.045860	0.000000	0.119703
8	0.046602	0.000000	0.121557
AVG	0.045916	0.000512	0.119991

Narzut proces a watek		
czas standardowy	czas CPU	czas zegarowy
308.14%	95.05%	524.54%

Wniosek:

Utworzenie procesu trwa ponad 5 razy więcej niż utworzenie wątku (kopiowanie całego programu, a nie jedynie tworzenie stosu, rejestru poleceń)

Analogiczny proces nastąpił dla funkcji clone() oraz dla kompilacji z optymalizacją.

Analiza wyników pokazała że optymalizacja nie ma większego wpływu na czas wykonywania, natomiast długość tworzenia nowego procesu (fork) jest około 5 razy większa niż długość tworzenia nowego wątku (clone). Wynika to zapewne z faktu że tworzeniu nowego procesu towarzyszy kopiowanie całego kodu procesu nadrzędnego, a dla wątku przydzielamy jedynie miejsce na stos i podajemy argumenty)

TWORZENIE WĄTKÓW I PROCESÓW. FUNKCJE clone() I fork().

clone() i fork()

fork() `pid_t fork(void)`
tworzy nowy proces i zwraca pid (proces id). Kopiuje kod programu i wykonuje się w tym samym miejscu. Aby sprawdzić w jakim procesie jesteśmy sprawdzamy pid (pid == 0, proces potomny)

```
pid_t wait(pid_t pid, int *status)
```

czeka na zakończenie procesu

clone() `int clone(int (*fn)(void *), void *child_stack, int flags, void *arg)`
tworzy nowy wątek.
Do funkcji clone podajemy: a) wskaźnik na funkcję od której zacznie się wywołanie wątku (można by nazwać ją funkcją wątku), b) wskaźnik na stos wątku (wcześniej utworzony), c) flagi, d) argumenty
Utworzony wątek nie zaczyna w tej samej linijce kodu w której został utworzony lecz wywołuje funkcję, którą mu podaliśmy. Zakończenie funkcji wątku prowadzi do zakończenia tego procesu (potomnego). Stos w procesorach obsługujacych systemach linux* idzie w dół więc wskaźnik stosu wskazuje jego najwyższy element.
*poza procesorami HP PA - ”(except the HP PA processors)”[6]

```
pid_t waitpid(pid_t pid, int *status, int options);
```

czeka na zakończenie wątku

Funkcję clone można skonfigurować dzięki flagom tak aby działała podobnie do fork.

Wielkość stosu:
Maksymalną wielkość stosu można zobaczyć poprzez polecenie ulimit -s (lub ulimit -a - wypisze wszystkie parametry). Domyślna wartość to 8192 kb.

Flagi w clone() [6]

CLONE_FILES
Odpowiada za kopiowanie deskryptorów plików (file descriptors). Skopiowanie oznacza że operacje odczytu lub tworzenia połączeń będą działały w oou procesach (ponieważ są współdzielone. BRAK flagi sprawia że fd's są kopiowane co sprawia, że np. zamknięcie pliku w jednym procesie nie wpływa na drugi ponieważ mają różne fd.

CLONE_FS
Współdzielenie informacji o systemie plików: "**root** of the filesystem, the **current working directory**, and the **umask**". Zmiany w jednym z procesów (funkcje: chroot, chdir, umask) wpływają na pozostałe. Brak flagi -> nie wpływają.

CLONE_IO (musi być flaga CONFIG_BLOCK) bo ma znaczenie tylko dla operacji blokowych.
Współdzielenie kontekstu IO. Planista systemowy traktuje je ten sam kontekst, dzielą ten sam czas dostępowy do IO. Przykład: jeżeli czytają z tego samego pliku to nie będą traktowane jak różne procesy i nie będą konkurować o czas dostępu do IO, co umożliwi lepsze wykorzystanie zasobów.

CLONE_VM
Współdzielenie pamięci. mmap() i munmap() działają na oba.

CLONE_SIGHAND od linux 2.6 wymagane też CLONE_VM
Współdzielenie signal handlers (sygnaly, np. przerwanie, zakończenie, błąd segmentacji itd.)

Kolizja w dostępie do współdzielonych danych
Jeżeli dwa wątki będą chciały w tym samym czasie uzyskać dostęp do wspólnej zmiennej (np. globalnej) to wystąpi tzw. kolizja. Na przykładzie z zajęć można było zaobserwować że zamiast oczekiwanej wartości 2000 zmienna globalna osiągała wartości typu 1300. Przy zastosowaniu optymalizacji podczas kompilacji zmienna osiągała wartość 2000, ponieważ inkrementowanie jej się nie wykonywało a od razu została przypisana wartość końcowa (2000).

źródła:

dostęp: 16.10.2025
[1] (materialy prof) http://ww1.metal.agh.edu.pl/~banas/PR/pomiar_czasu.pdf
[2] (clock) https://en.cppreference.com/w/c/chrono/clock.html
[3] (getrusage) https://man7.org/linux/man-pages/man2/getrusage.2.html
[4] (getrusage)
https://pubs.opengroup.org/onlinepubs/009696699/functions/getrusage.html
[5] (settimeofday) https://man7.org/linux/man-pages/man2/settimeofday.2.html
[6] (clone) https://man7.org/linux/man-pages/man2/clone.2.html