

Systemy operacyjne – zestaw 1

Optymalizacja

Optymalizacja

(<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>)

- -O – kompilator próbuje optymalizować kod i czas wykonania, bez wykonywania operacji znacząco zwiększających czas kompilacji
- -O1 – możliwy dłuższy czas kompilacji, zużywa dużo pamięci dla dużych funkcji
- -O2 – używane jeszcze więcej opcji kompilacji, które nie powodują zwiększenia pamięci programu, w porównaniu do -O zwiększa czas kompilacji i wydajność kodu, używa wszystkich flag używanych przez -O a także dodatkowe flagi
- -O3 – używa wszystkie flagi używane przez -O2, a także dodatkowe flagi
- -O0 – ogranicza czas kompilacji, wartość domyślna
- -Os – optymalizacja rozmiaru, używa wszystkie flagi -O2, które nie zwiększają kodu programu, używa też dodatkowych flag zmniejszających rozmiar kodu

Pomiar czasu

Funkcje czekania

```
#include <unistd.h>
```

```
unsigned int sleep(unsigned int seconds) ;
```

```
#include <time.h>
```

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

```
struct timespec {
```

```
    time_t tv_sec; /* seconds */
```

```
    long tv_nsec; /* nanoseconds */
```

```
};
```

Zegary POSIX

- Typ danych `clock_t` – reprezentuje takty zegara
- Typ danych `clockid_t` – reprezentuje określony zegar Posix
- Są 4 rodzaje zegarów – zalecany to `CLOCK_REALTIME` – ogólnosystemowy zegar czasu rzeczywistego
- `#include <time.h>`
- `int clock_getres(clockid_t clk_id, struct timespec *res)` – odczytuje rozdzielczość zegara wyspecyfikowanego w parametrze *clk_id*

Pomiar czasu (1)

- `#include <time.h>`
`int clock_gettime(clockid_t clk_id, struct timespec *tp)` – pobranie wartości zegara
`int clock_settime(clockid_t clk_id, const struct timespec *tp)` - ustawienie wartości zegara

Pomiar czasu (2)

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buffer);
```

Pola struktury tms

tms_utime – czas cpu wykonywania procesu w trybie użytkownika

tms_stime – czas cpu wykonywania procesu w trybie jądra

tms_cutime – suma czasów cpu wykonywania procesu i wszystkich jego potomków w trybie użytkownika

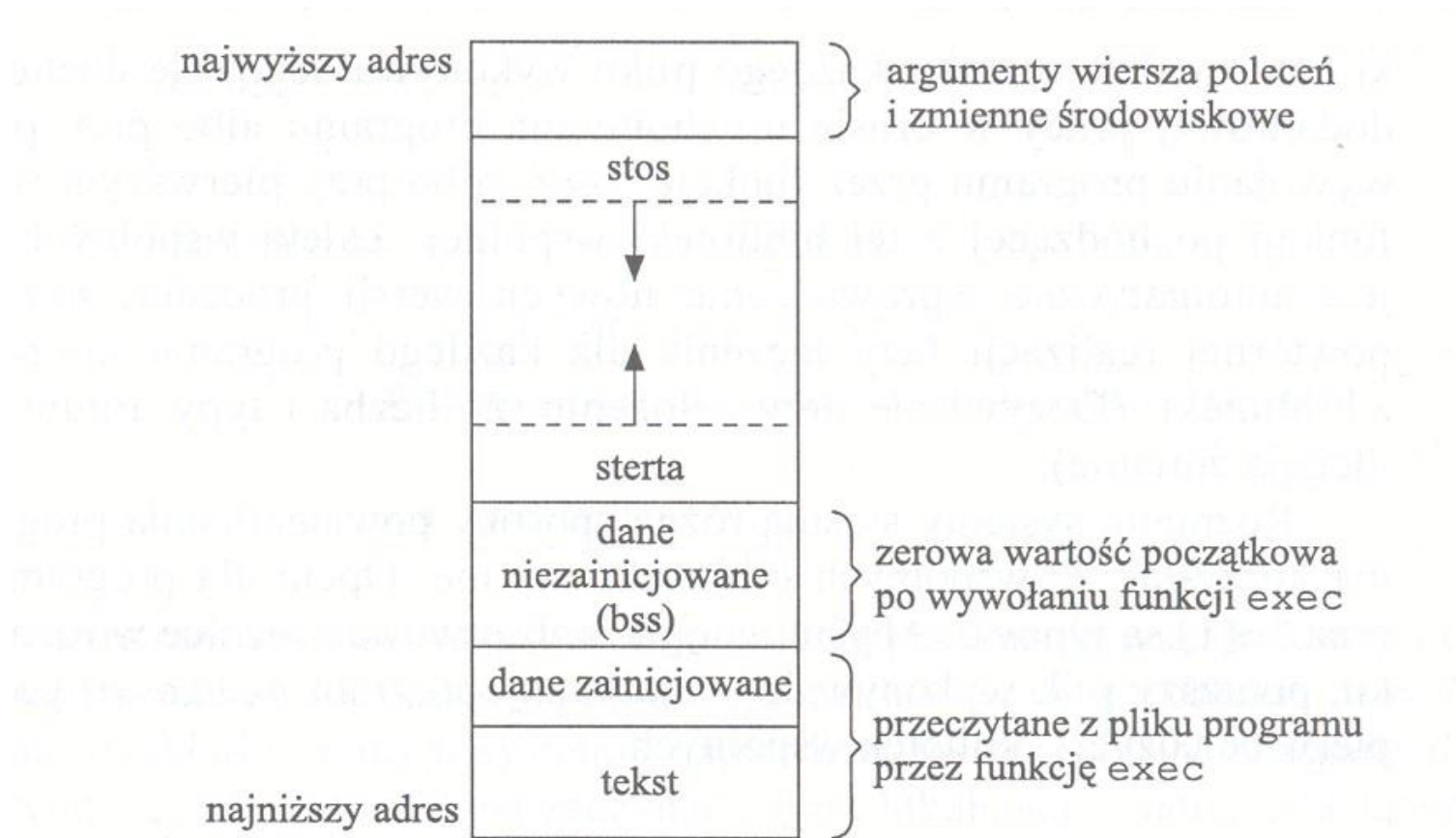
tms_cstime - suma czasów cpu wykonywania procesu i wszystkich jego potomków w trybie jądra

Bibliografia

- Robert Love, „Linux Kernel Development. A thorough guide to the design and implementation of the Linux kernel”, Novel Press, Third edition, 2010

Zarządzanie pamięcią

Typowa organizacja pamięci w procesie



Unix. Funkcje do alokacji pamięci dynamicznej w programach

Alokacja pamięci: (standard ANSI C)

- **malloc** – alokuje w pamięci wskazaną liczbę bajtów. Wartość początkowa zawartości pamięci nie jest określona
- **calloc** – alokuje przestrzeń dla określonej liczby obiektów o zadanym obszarze. Cały zarezerwowany obszar jest wypełniony bitami zerowymi
- **realloc** – zmienia rozmiar poprzednio zaalokowanego obszaru (zwiększa go lub zmniejsza). Jeśli rozmiar rośnie, może to oznaczać przesunięcie wcześniej zaalokowanego obszaru w inne miejsce, aby dodać wolną przestrzeń na jego końcu. W takiej sytuacji nie jest określona wartość początkowa fragmentu pamięci między końcem starego a końcem nowego obszaru.
- **free** – zwalnia pamięć wskazaną przez ptr

```
#include <stdlib.h>
void * malloc(size_t size);
void * calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
void free(void *ptr);
```

Unix. Funkcje do zarządzania pamięcią.

- Funkcje alokujące są na ogół implementowane za pomocą funkcji systemowej **sbrk(2)**, które rozszerza lub zawęża stertę procesu
- Choć wywołanie funkcji sbrk może rozszerzyć lub zawęzić pamięć procesu, to jednak większość wersji funkcji malloc i free nigdy nie zmniejsza rozmiaru pamięci procesu – zwalniana pamięć staje się dostępna dla kolejnych alokacji, ale nie powraca do jądra systemu – jest utrzymywana w puli, którą dysponuje funkcja malloc.
- Uwaga: większość implementacji alokuje nieco więcej pamięci, niż jest to wymagane, dodatkowy obszar jest używany do przechowywania specjalnych danych jak: rozmiar alokowanego bloku, wskaźnika do kolejnego bloku do alokacji.

Inne funkcje mechanizmu alokacji: mallinfo

- mallinfo – dostarcza charakterystyki mechanizmu alokacji:
- struct mallinfo mallinfo(void)
- unsigned long arena; //total space
- unsigned long ordblks; //number of ordinary blocks
- unsigned long smblks; //number of small blocks
- unsigned long hblkhd; //space in holding block headres
- unsigned long hblks; //number of holding blocks
- unsigned long usmblks; //space in small blocks in use
- unsigned long fsmblks; //space in free small blocks
- unsigned long uordblks; //space in ordinary blocks in use
- undigned long fordblks; //space in free ordinary blocks
- unsigned lon keepcostl //space penalty if keep option is used

Użycie obszarów pamięci (1)

- Przykładowy program:
`int main(int argc, char * argv[])`
- `{`
 - `return 0;`
- `}`
- Proces zawiera obszary odpowiadające sekcjom tekstu, danych i bss
- Zakładając, że proces jest dynamicznie zlinkowany z biblioteką C, analogiczne trzy obszary pamięci istnieją także dla `libc.so` (biblioteka c) oraz dla `ld.so` (linkera dynamicznego)
- Proces posiada także obszar pamięci odpowiadający za stos
- Poniższe dane w pliku `/proc/<pid>/maps` przedstawiają obszary pamięci, mają postać:
początek obszaru-koniec obszaru prawa dpstępu duży:mały iwęzeł plik

```
rlove@wolf:~$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530      /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530      /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029      /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029      /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276       /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276       /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bffffe00-c0000000 rwxp fffff000 00:00 0
```

Użycie obszarów pamięci (2)

- Pierwsze trzy wiersze, to sekcja tekstu, danych i bss biblioteki C (libc.so)
- Następne dwa wiersze to sekcje kodu i danych programu wykonywalnego
- Następne trzy wiersze to sekcja tekstu, danych i bss linkera dynamicznego (ld.so)
- Ostatni wiersz to obszar stosu
- Cała przestrzeń adresowa zajmuje około 1340KB, ale tylko 40KB są zapisywalne i prywatne
- Jeśli obszar pamięci jest współdzielony lub niemodyfikowalny, jądro przechowuje tylko jedną jego kopię w pamięci
 - Dlatego biblioteka C potrzebuje tylko 1212KB pamięci fizycznej dla wszystkich procesów
- Obszary pamięci bez zmapowanego pliku i o i-węźle 0 – są to strony zerowe (zero page): mapowania zawierające tylko zera
 - Przez zmapowanie strony zerowej na zapisywalne obszary pamięci, obszar jest "inicjalizowany" zerami, co jest oczekiwane dla bss

```
rlove@wolf:~$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530      /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530      /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029      /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029      /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276       /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276       /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bffffe000-c0000000 rwxp fffff000 00:00 0
```


Bibliografia

- Robert Love, „Linux Kernel Development. A thorough guide to the design and implementation of the Linux kernel”, Novel Press, Third edition, 2010

Biblioteke

Biblioteki (Libraries)

- Co to są biblioteki?

Biblioteka jest zbiorem implementacji zachowań, opisanych w języku programowania, która ma dobrze zdefiniowany interfejs, przez który zachowania są wywoływane [Wikipedia]

"program library" jest plikiem zawierającym skompilowany kod i dane, które będą włączone potem do programu/programów, umożliwiają modularne programowanie, szybszą rekompilację i łatwiejsze uaktualnienia [The Linux Documentation Project]

Libraries

- Biblioteki można podzielić na trzy rodzaje: : statyczne, współdzielone i dynamicznie ładowane
- Styczne biblioteki są dołączane do programu wykonywalnego przed jego uruchomieniem
- Współdzielone biblioteki są ładowane w momencie uruchomienia programu i mogą być współdzielone z innymi programami
- Dynamicznie ładowane biblioteki są ładowane, gdy program wykonywalny się wykonuje.

Biblioteki statyczne (Static Libraries)

- Biblioteki statyczne są zbiorami plików obiektowych. Zazwyczaj mają rozszerzenie ".a" .
- Biblioteki statyczne pozwalają użytkownikom linkować się do plików obiektowych bez rekompilacji kodu. Pozwalają także dystrybuować biblioteki bez rozpowszechniania kodu źródłowego.

Biblioteki statyczne

•Przykłady:

•**my_library.h**

•#pragma once

•namespace my_library {

• extern "C" void my_library_function();

•}

•**my_library.c**

...

•#include "my_library.h"

•void my_library_function() {

...

•}

Biblioteki statyczne

- Przykład:

- main.c**

- `#include "my_library.h"`

- `int main() {
 my_library_function();`

- `}`

Biblioteki statyczne

- Przykład – kompilacja z plikami obiektowymi

- \$ gcc -c my_library.c

- \$ gcc -c main.c

- \$ gcc main.o my_library.o -o main

- \$./main

Biblioteki statyczne

- Przykłady – kompilacja jako biblioteka statyczna

- \$ gcc -c my_library.c

- \$ ar rcs libmy_library.a my_library.o

- \$ gcc -c main.c

- \$ gcc main.o libmy_library.a -o main

- \$./main

- ...

- \$ gcc main.o -l my_library -L ./ -o main

- \$./main

Biblioteki współdzielone (Shared Libraries)

- Biblioteki współdzielone są ładowane gdy program jest ładowany. Wszystkie programy mogą współdzielić dostęp do współdzielonych bibliotek i będzie uaktualniona (upgraded) jeśli nowa wersja biblioteki zostanie zainstalowana
- Może być zainstalowanych wiele wersji bibliotek, by pozwolić programom ze specyficznymi potrzebami na używanie konkretnej wersji biblioteki
- Biblioteki te mają zazwyczaj rozszerzenie **.so**

Biblioteki współdzielone

- Biblioteki współdzielone używają specyficznej reguły nazewnictwa
- Każda biblioteka ma "**soname**" zaczynające się do prefiksu "**lib**", po których następuje nazwa (**name**) biblioteki, po czym rozszerzenie **".so"** ta następnie kropkę i wersję biblioteki (**version numer**).

Biblioteki współdzielone

- Każda biblioteka ma także nazwę rzeczywistą "real name" – nazwę pliku z kodem biblioteki. Rzeczywista nazwa "real name" obejmuje "soname", kropkę, **version** number i opcjonalnie kropkę i **release number**.
- Oba numery (version i release) umożliwiają wybór dokładnej wersji biblioteki.

Biblioteki współdzielone

- Dla jednej biblioteki współdzielonej system często ma wiele linków wskazujących na tę samą nazwę

- Przykład:

- **soname**

- /usr/lib/libreadline.so.3

- **Linkowanie do realname :**

- /usr/lib/libreadline.so.3.0

- **Lub:**

- /usr/lib/libreadline.so

- **Linkowanie do:**

- /usr/lib/libreadline.so.3

Biblioteki współdzielone

- Przykłady – Kompilowanie biblioteki współdzielonej

- `$ gcc -fPIC -c my_library.c`

- `(-fPIC position independent code, potrzebny do kodu biblioteki współdzielonej)`

- `$ gcc -shared -Wl,-soname,libmy_library.so.1 \`

- `-o libmy_library.so.1.0.1 my_library.o -lc`

- `$ ln -s libmy_library.so.1.0.1 libmy_library.so.1`

- `$ ln -s libmy_library.so.1 libmy_library.so`

Biblioteki współdzielone

- Przykład – użycie biblioteki współdzielonej

- `$ gcc main.c -lmy_library -L ./ -o main`

- `$./main`

Biblioteki współdzielone

- Problemem jest, że mamy stałą ścieżkę do biblioteki
- Biblioteka musi być w tym samym katalogu
- Położenie bibliotek współdzielonych może się różnić w zależności do systemu.

Biblioteki współdzielone

- `$LD_LIBRARY_PATH` służy do ustawienia ścieżki poszukiwań bibliotek współdzielonych
- Kiedy program wymagający bibliotek współdzielonych jest uruchomiony, system poszukuje ich w katalogach podanych w `$LD_LIBRARY_PATH`.

Biblioteki współdzielone

- Jeśli chcesz zainstalować swoją bibliotekę w systemie, możesz skopiować pliki .so do jednej ze standardowych katalogów - `/usr/lib` i wywołać `ldconfig`
- Każdy program używający biblioteki może teraz odwołać się do niej poprzez `-lmy_library` i system znajdzie ją w `/usr/lib`

Biblioteki ładowane dynamicznie (Dynamically Loaded Libraries)

- Dynamicznie ładowane biblioteki są ładowane przez sam program z poziomu kodu źródłowego.
- Biblioteki są zbudowane jako standardowe obiekty lub biblioteki współdzielone, jedyną różnicą jest to, że biblioteki nie są ładowane podczas fazy linkowania przy kompilacji lub uruchomienia, ale w punkcie ustalonym przez programistę.

Biblioteki ładowane dynamicznie

- Funkcje odpowiedzialne za operacje na bibliotekach ładowanych dynamicznie:

- `void* dlopen(const char *filename, int flag);` – Otwiera bibliotekę, przygotowuje ją do użycia i zwraca wskaźnik/uchwyt na bibliotekę.

- `void* dlsym(void *handle, char *symbol);` – Przegląda bibliotekę szukając specyficznego symbolu.

- `void dlclose();` – Zamyka bibliotekę .

Biblioteki ładowane dynamicznie

```
•#include <dlfcn.h>
```

```
int main() {
```

```
    void *handle = dlopen("libmy_library.so", RTLD_LAZY);  
    if(!handle){/*error*/}
```

```
    void (*lib_fun)();  
    lib_fun = (void (*)())dlsym(handle, "my_library_function");  
    if(dlerror() != NULL){/*error*/}
```

```
    (*lib_fun)();
```

```
    dlclose(handle);
```

```
}
```

GNU Make

Make tool and Makefile

Co to jest Make?

- Narzędzie generacji kodu wykonywalnego
- Uwzględnia modyfikacje plików źródłowych

Możliwości

- Automatycznie określa, które pliki potrzebują uaktualnienia
- Nie ograniczony do szczególnego języka
- Użytkownik końcowy może zbudować i zainstalować pakiet bez znajomości szczegółów, jak to przeprowadzić

Reguły (Rules) i Cele (Targets)

- Reguła (rule) w pliku makefile mówi użytkownikowi jak wykonywać serie poleceń by zbudować plik docelowy z plików źródłowych
- Określa listę zależności dla pliku docelowego

```
target:    dependencies ...  
           commands  
           ...
```

Budowa Procesu

- Kompilator pobiera pliki źródłowe i wyjściowe pliki obiektowe (object files).
- Linker pobiera pliki obiektowe i tworzy plik wykonywalny

Prosty Makefile

```
all: hello

hello: main.o factorial.o hello.o
    gcc main.o factorial.o hello.o -o hello

main.o: main.c
    gcc -c main.c

factorial.o: factorial.c
    gcc -c factorial.c

hello.o: hello.cpp
    gcc -c hello.c

clean:
    rm *o hello
```

Prosty Makefile ze zmiennymi

```
CC=gcc
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.c
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.c
    $(CC) $(CFLAGS) factorial.c

hello.o: hello.c
    $(CC) $(CFLAGS) hello.c

clean:
    rm *o hello
```

Użycie Makefile

Wykonanie makefile

```
make
```

Konkretny cel (target) może być wykonany przez:

```
make target_label
```

Na przykład, wykonanie polecenia *rm* z poprzedniego slajdu:

```
make clean
```

GDB:GNU Debugger

Co to jest GDB?

- GDB, GNU debugger, które pozwalają na obserwację wykonania programu w danych punktach wykonania, lub co było robione w momencie, gdy nastąpiło załamanie się/błąd programu
- Debuggowany program może być napisany w różnych językach programowania - C, C++, Objective-C, Pascal (i wielu innych).

Zastosowania GDB

- Uruchom program i wyspecyfikuj działania wpływające na jego zachowanie
- Zatrzymaj program, jeśli odpowiedni warunek jest spełniony
- Sprawdź, co się stało, gdy program się zatrzymał
- Zmodyfikuj program tak, że następuje poprawa efektów jednego z błędów i szukaj ewentualnych innych błędów

Kompilacja programu w trybie do debugowania przy pomocy GDB

- Normalna kompilacja programu

```
gcc [flags] <source files> -o <output file>
```

- Kompilacja włączająca wsparcie dla debugowania poprzez dodanie symbolu `-g`

```
gcc [other flags] -g <source files> -o <output  
file>
```

Uruchamianie GDB

- Po napisaniu „**gdb**” lub „**gdb my_prog.x**”, debugger zgłosi się następująco:
 - **(gdb)**
- Jeżeli debuggowany program nie został wyspecyfikowany podczas uruchomienia gdb, można go wskazać używając polecenia „**file**”:
 - **(gdb) file my_prog.x**

Uruchomienie programu

- Po załadowaniu programu do debuggera, można program uruchomić w debuggerze przy pomocy polecenia „**run**”:
 - **(gdb) run**
- Jeśli nie wystąpią żadne błędy podczas wykonania, program zakończy się bez dodatkowych informacji. Jeśli pojawią się problemy, wyświetlona zostanie odpowiednia informacja, gdzie nastąpił błąd i program się załamał (gdzie nastąpił „crash”);

```
Program received signal SIGSEGV, Segmentation fault.  
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,  
r2=4, c2=6) at sum-array-region2.c:12
```

```

"infinite.c" 14L, 177C written
[root@aninath-lnx GDB]# gcc -g -Wall -o infinite infinite.c
infinite.c: In function âmainâ:
infinite.c:6: warning: division by zero
[root@aninath-lnx GDB]# ./infinite
Floating point exception (core dumped)
[root@aninath-lnx GDB]# ls -ltr
total 92
-rw-r--r-- 1 root root    177 May 17 23:59 infinite.c
-rwxr-xr-x 1 root root  8343 May 17 23:59 infinite
-rw----- 1 root root 155648 May 17 23:59 core.infinite.21364
[root@aninath-lnx GDB]# gdb infinite core.infinite.21364
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./infinite'.
Program terminated with signal 8, Arithmetic exception.
[New process 21364]
#0  0x00000000004004f6 in main () at infinite.c:6
6      i=i/0;
(gdb) bt
#0  0x00000000004004f6 in main () at infinite.c:6
(gdb)

```

Obserwacja błędów
przy wykorzystaniu
pliku zrzuconego
obrazu pamięci
(pliku **core**)

Breakpoints

- Można użyć polecenia **break**, aby zatrzymać program w wybranym punkcie:
 - (gdb) **break my_file.c:7** → ustaw breakpoint w linii 7 w pliku **my_file.c**
- Można także ustawić breakpoint w wybranej funkcji:
 - (gdb)**break myfunc** → *myfunc* jest nazwą funkcji w programie
- Można także ustawić warunkowe breakpoints, które umożliwiają pominięcie niepotrzebnych kroków wykonania:
 - (gdb) **break file1.c:6 if i >= ARRAYSIZE** → ustawia **break point** jeśli wartość **i** jest większa lub równa wartości **ARRAYSIZE**

Kontynuacja po Breakpoint

- Po ustawieniu breakpoint, można uruchomić program (**run**) ponownie. Możliwe są także następujące działania:

Można napisać **continue** by dostać się do następnego breakpoint'a

- **(gdb)continue**

Można napisać **step** by wykonać pojedynczy wiersz

- **(gdb)step**

Można napisać **next**, by znów wykonać pojedynczy krok, ale nie wykonuje on wszystkich wierszy podprocedury;

- **(gdb)next**

Przykład: różnica między **next** i **step**

- Mamy część kodu i przyjmujemy, że znajdujemy się w linii *foo()* znajdującej się w funkcji *int main*.

```
void foo() {  
    for ( int i = 0; i < v.size(); i++ ) {  
        ...  
    }  
}  
  
int main() {  
    foo();  
}
```

- Jeżeli napiszemy **next**, przemieścimy się do następnej linii, którą jest „}” w funkcji *int main()*.
- Jeżeli napiszemy **step**; przemieścimy się do pętli for wewnątrz funkcji *foo()*.

Zmienne

- Jeżeli chcemy obejrzeć wartości zmiennych podczas debuggowania, można użyć polecenia **print**;
 - `(gdb) print my_var`
 - `(gdb) print/x my_var` → dla wyświetlenia w postaci szesnastkowej

Debuggowanie wykonującego się procesu

```
#include <stdio.h>
#include <unistd.h>

static void PrintMessage(int i);
static void GoToSleep(void);

int main(void)
{
    int i = 100000;

    while ( 1 )
    {
        PrintMessage( i );
        GoToSleep();
        i -= 1;
    }

    return 0;
}

void PrintMessage(int i)
{
    printf("%d bottles of beer on the wall.\n", i);
}

static void GoToSleep(void)
{
    sleep(3);
}
```

Debuggowanie wykonującego się procesu (cd)

```
$ ./beer-process &  
[1] 17399  
p@satan$ 100000 bottles of beer on the wall.  
99999 bottles of beer on the wall.  
99998 bottles of beer on the wall.  
99997 bottles of beer on the wall.
```

```
$ gdb beer-process 17399  
Attaching to program: code/running_process/beer-process, process 17399  
0x410c64fb in nanosleep () from /lib/tls/libc.so.6  
(gdb)
```

In another console



```
(gdb) quit  
The program is running.  Quit anyway (and detach it)? (y or n) y  
Detaching from program: code/running_process/beer-process,  
process 17399
```

```
(gdb) bt
#0  0x410c64fb in nanosleep () from /lib/tls/libc.so.6
#1  0x410c6358 in sleep () from /lib/tls/libc.so.6
#2  0x0804841f in GoToSleep () at beer-process.c:32
#3  0x080483e0 in main () at beer-process.c:14
```

```
(gdb) frame 3
#3  0x080483eb in main () at beer-process.c:15
15          GoToSleep();
(gdb) print i
$1 = 99997
```

```
(gdb) frame 3
#3  0x080483eb in main () at beer-process.c:15
15          GoToSleep();
(gdb) set var i = 99999999
```

```
(gdb) next
Single stepping until exit from function nanosleep,
which has no line number information.
0x410c6358 in sleep () from /lib/tls/libc.so.6
(gdb) step
Single stepping until exit from function sleep,
which has no line number information.
GoToSleep () at beer-process.c:34
34      }
(gdb) bt
#0  GoToSleep () at beer-process.c:34
#1  0x080483eb in main () at beer-process.c:15
```

```
(gdb) quit
The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: code/running_process/beer-process,
process 17399
```

Debuggowanie wykonującego się procesu(cd)

- Z wykorzystaniem polecenia **attach**.

```
$ gdb
(gdb) attach 17399
Attaching to process 17399
Reading symbols from code/running_process/beer-process...done.
0x410c64fb in nanosleep () from /lib/tls/libc.so.6
(gdb)
```

- Po zakończeniu można odłączyć debugger od procesu (**detach**).

Inne użyteczne polecenia

- **backtrace** – wyświetl ścieżkę stosu wywoływanej funkcji
- **finish** – wykonuj się do zakończenia aktualnej funkcji
- **delete** – usuń podany breakpoint
- **info breakpoints** - wyświetl informacje o wszystkich ustawionych breakpoints
- **help** – można używać wraz z nazwą wybranego polecenia lub bez niej, powoduje wyświetlenie listy dostępnych poleceń lub opis działania wybranego polecenia

Bibliografia

- University of Maryland Computer Science Department Tutorials
- Tutorialspoint website
- Official website of GNU Operating System