1. Напишіть програму для визначення моменту, коли time\_t закінчиться.Дослідіть, які зміни відбуваються в залежності від 32- та 64-бітної архітектури. Дослідіть сегменти виконуваного файлу.

#include <stdio.h>

#include <time.h>

#include <limits.h>

#include <stdint.h>

int main(){

    time\_t time = 0;

    while(time >= 0) time++;

    printf ("max value of time\_t -> %ld /n", (long)time);

    printf ("represent -> %s /n", ctime (&time));

    printf ("time\_t size -> %lu bytes /n", sizeof (time\_t));

    printf ("system bit rate -› ", sizeof (void \*)\*8);

    return 0;

}

#include <stdio.h>

#include <limits.h>

#include <time.h>

#include <stdint.h>

int main(){

    time\_t t\_max = (time\_t)LONG\_MAX;

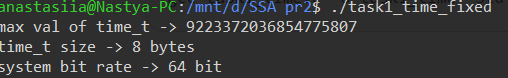
    printf ("max val of time\_t -> %ld \n", (long)t\_max);

    printf ("time\_t size -> %lu bytes \n", sizeof (time\_t));

    printf ("system bit rate -> %lu bit \n", sizeof (void \*)\*8);

    return 0;

}



При запуску програма може виконуватись дуже довго — усе залежить від системи. У 32-бітній системі time\_t обмежений значенням 2^31 - 1, тому цикл завершується швидше. У 64-бітній — це 2^63 - 1, що дорівнює сотням мільярдів років, тож чекати недоцільно. Краще одразу задати максимальне значення вручну.

Завдання 2.2.

Розгляньте сегменти у виконуваному файлі.

1. Скомпілюйте програму &quot;hello world&quot;, запустіть ls -l для виконуваного файлу, щоб отримати його загальний розмір, і запустіть size, щоб отримати розміри сегментів всередині нього.

2. Додайте оголошення глобального масиву із 1000 int, перекомпілюйте й повторіть вимірювання. Зверніть увагу на відмінності.

3. Тепер додайте початкове значення в оголошення масиву (пам’ятайте, що C не змушує вас вказувати значення для кожного елемента масиву в ініціалізаторі). Це перемістить масив із сегмента BSS у сегмент даних. Повторіть вимірювання. Зверніть увагу на різницю.

4. Тепер додайте оголошення великого масиву в локальну функцію. Оголосіть другий великий локальний масив з ініціалізатором. Повторіть вимірювання. Дані розташовуються всередині функцій, залишаючись у виконуваному файлі? Яка різниця, якщо масив ініціалізований чи ні?

5. Які зміни відбуваються з розмірами файлів і сегментів, якщо ви компілюєте для налагодження? Для максимальної оптимізації?

Проаналізуйте результати, щоб переконатися, що:

● сегмент даних зберігається у виконуваному файлі;

● сегмент BSS не зберігається у виконуваному файлі (за винятком

примітки щодо його вимог до розміру часу виконання);

● текстовий сегмент більшою мірою піддається перевіркам

оптимізації;

● на розмір файлу a.out впливає компіляція для налагодження, але не

сегменти.

#include <stdio.h>

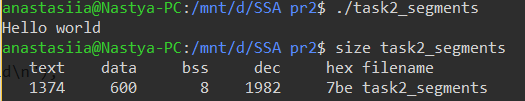
//v1

int main(){

printf ("Hello world\n");

return 0;

}



Базова програма виконується коректно.

//v2

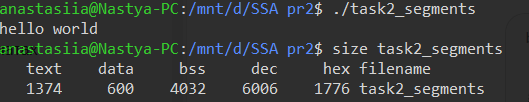
int arr [1000];

int main(){

  printf ("hello world\n");

  return 0;

}



Бачимо, що секція .bss збільшилась, оскільки array є неініціалізованою глобальною змінною.

//v3

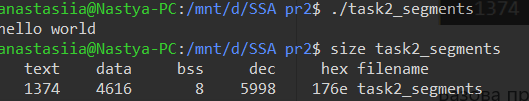
int arr [1000] = {5};

int main(){

  printf ("hello world\n");

  return 0;

}



Секція .bss зменшилась, а .data збільшилась, оскільки масив тепер має початкове значення.

//v4

 void local\_arr\_function(){

      int arr[1000];

     static int s\_arr [1000] = {9};

 }

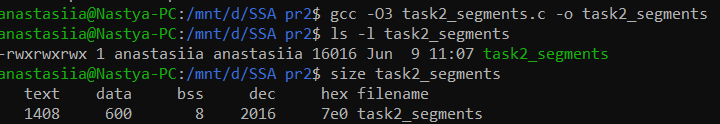
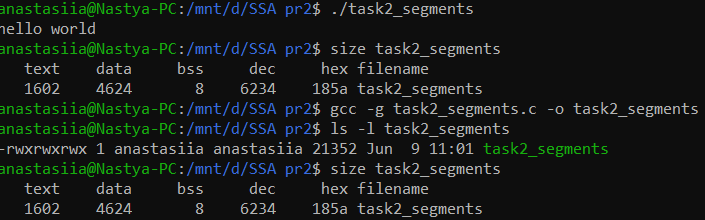
 int main(){

     printf ("hello world\n");

     local\_arr\_function();

     return 0;

 }



Змінна arr1 розміщується в стеку, тому її розмір **не впливає** на розмір виконуваного файлу. Натомість arr2, оголошена як static, потрапляє в сегмент .data, і **впливає** на розмір файлу.

Якщо зібрати програму з опцією **-g** (для налагодження), **розмір файлу не змінюється суттєво**. Але при компіляції з **максимальною оптимізацією (-O3)**, видно зменшення розміру — компілятор застосовує агресивні методи оптимізації, що дозволяють скоротити зайві дані.

Завдання 2.3.

Скомпілюйте й запустіть тестову програму, щоб визначити приблизне розташування стека у вашій системі:

#include &lt;stdio.h&gt;

int main() {

int i;

printf(&quot;The stack top is near %p\n&quot;, &amp;i);

return 0;

}

Знайдіть розташування сегментів даних і тексту, а також купи всередині сегмента даних, оголосіть змінні, які будуть поміщені в ці сегменти, і виведіть їхні адреси. Збільшіть розмір стека, викликавши функцію й оголосивши кілька великих локальних масивів. Яка зараз адреса вершини стека?

Примітка: стек може розташовуватися за різними адресами на різних архітектурах та різних ОС. Хоча ми говоримо про вершину стека, на більшості процесорів стек зростає вниз, до пам’яті з меншими значеннями адрес.

#include <stdio.h>

#include <stdlib.h>

int data = 0;

int bss;

void increase\_stack\_top();

void text(){}

int main(){

    int local;

    int i; // var for stacks top tracking

    char \*heap = (char \*)malloc (10);

    if (!heap){

        return 1;

    }

    printf ("local →> %p\n", (void \*)&local);

    printf ("text -> %p\n",(void \*)&text);

    printf ("data -> %p",  (void \*)&data);

    printf ("bss -> %p\n", (void \*)&bss);

    printf ("mallocs memory address -> %p\n", (void \*)heap);

    printf ("stack top address -> %p\n", &i);

    increase\_stack\_top();

    free (heap);

    return 0;

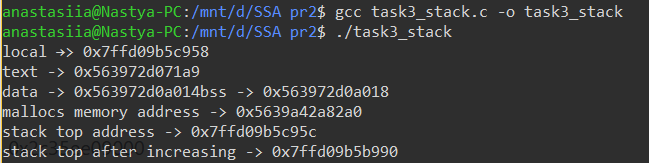
}

void increase\_stack\_top(){

    int arr [1000];

    printf ("stack top after increasing -> %p\n",&arr );

}



Завдання 2.4

Ваше завдання – дослідити стек процесу або пригадати, як це робиться. Ви можете:

● Автоматично за допомогою утиліти gstack.

● Вручну за допомогою налагоджувача GDB.

Користувачі Ubuntu можуть зіткнутися з проблемою: на момент написання (Ubuntu 18.04) gstack, схоже, не був доступний (альтернативою може бути pstack). Якщо gstack не працює, використовуйте другий метод – через GDB, як показано нижче.

Спочатку подивіться на стек за допомогою gstack(1). Нижче наведений приклад стека bash (аргументом команди є PID процесу):

$ gstack 14654

#0 0x00007f359ec7ee7a in waitpid () from /lib64/libc.so.6

#1 0x000056474b4b41d9 in waitchild.isra ()

#2 0x000056474b4b595d in wait\_for ()

#3 0x000056474b4a5033 in execute\_command\_internal ()

#4 0x000056474b4a5c22 in execute\_command ()

#5 0x000056474b48f252 in reader\_loop ()

#6 0x000056474b48dd32 in main ()

$

Розбір стека:

● Номер кадру стека відображається ліворуч перед символом #.

● Кадр #0 – це найнижчий кадр. Читайте стек знизу вверх (тобто від

main() – кадр #6 – до waitpid() – кадр #0).

● Якщо процес багатопотоковий, gstack покаже стек кожного потоку

окремо.

Аналіз стека в режимі користувача через GDB

Щоб переглянути стек процесу вручну, використовуйте GDB, приєднавшись до процесу. Нижче наведена невелика тестова програма на C, що виконує кілька вкладених викликів функцій. Граф викликів виглядає так: main() --&gt; foo() --&gt; bar() --&gt; bar\_is\_now\_closed() --&gt; pause() Системний виклик pause() – це приклад блокуючого виклику. Він переводить викликаючий процес у сплячий режим, очікуючи (або блокуючи) сигнал. У цьому випадку процес блокується, поки не отримає будь-який сигнал.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#define MSG "In function %20s; &amp;localvar = %p\n"

static void bar\_is\_now\_closed(void) {

    int localvar = 5;

    printf(MSG, \_\_FUNCTION\_\_, &localvar);

    printf("\n Now blocking on pause()...\n");

    pause();

}

static void bar(void) {

    int localvar = 5;

    printf(MSG, \_\_FUNCTION\_\_, &localvar);

    bar\_is\_now\_closed();

}

static void foo(void) {

    int localvar = 5;

    printf(MSG, \_\_FUNCTION\_\_, &localvar);

    bar();

}

int main(int argc, char \*\*argv) {

    int localvar = 5;

    printf(MSG, \_\_FUNCTION\_\_, &localvar);

    foo();

    exit(EXIT\_SUCCESS);

}

У іншому терміналі вводимо команду для відслідковування PID

$ pgrep t4 1605

1605 - PID, за яким ми відслідкуємо процес через gdb

(gdb) bt #0 0х000000082228777а in \_sigsuspend () from /lib/libc.so.7 #1 0x00000008221f dc35 in pause () from /lib/libc.so.7 #2 0x0000000000400686 in bar\_is\_now\_closed () #3 0x00000000004006bd in bar () #4 0x00000000004006ef in foo () #5 0х0000000000400728 in main ()

Завдання 2.5

Відомо, що при виклику процедур і поверненні з них процесор використовує стек.Чи можна в такій схемі обійтися без лічильника команд (IP), використовуючи замість нього вершину стека? Обґрунтуйте свою відповідь та наведіть приклади.

21. Використайте valgrind для профілювання пам’яті.

#include <stdlib.h>

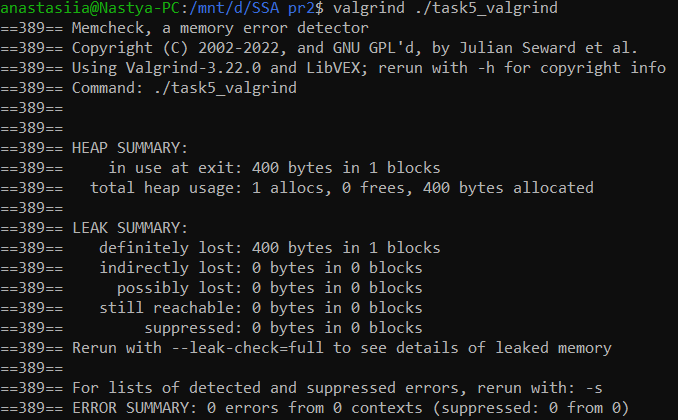
int main() {

    int \*arr = malloc(100 \* sizeof(int));

    arr[0] = 123;

    return 0;

}



**Чи можна обійтись без лічильника команд (IP), використовуючи лише вершину стека?**

**Ні, не можна.**

**Обґрунтування:**  
Лічильник команд (Instruction Pointer / Program Counter) є ключовим елементом керування потоком виконання програми. Він зберігає **адресу наступної інструкції**, яку процесор має виконати. Навіть коли ми використовуємо стек для збереження адрес повернення з функцій (наприклад, у call в x86), сам IP все одно потрібен для:

* Переходу між інструкціями.
* Визначення адреси при виклику функцій, умовних переходах, циклах.
* Відновлення нормального виконання після повернення з процедур.

**Стек лише тимчасово зберігає IP**, коли ми переходимо до іншої функції, але виконання все одно потребує IP.

Стек і IP доповнюють одне одного, але один не може повністю замінити інший.