Projekt 2: Zarządzanie pamięcią

Materiały wykładowe dostępne w usłudze Pliki w Chmurze Politechniki Łódzkiej: (https://pwc.p.lodz.pl/d/60cd5c043f12486990a7/) http://bit.ly/sysopy2. Oficjalne repozytorium przedmiotu znajduje się w usłudze GitHub: https://github.com/tomekjaworski/SO2

> Jeżeli w tekście są braki lub niejasności to skontaktuj się z koordynatorem przedmiotu (Tomasz Jaworski, tjaworski@iis.p.lodz.pl) w celu korekty instrukcji.

Zadanie

Celem projektu jest przygotowanie managera pamięci do zarządzania stertą własnego programu. W tym celu należy przygotować własne wersje funkcji `malloc`, `calloc`, `free`, `realloc` wraz z dodatkowymi funkcjami narzędziowymi, pozwalającymi na monitorowania stanu, spójności oraz defragmentacji obszaru sterty.

Przygotowane funkcje muszą realizować następujące funkcjonalności:

- Standardowe zadania alokacji/dealokacji zgodne z malloc API. Należy **dokładnie** odwzorować zachowanie własnych implementacji z punktu widzenia wywołującego je kodu.
- Możliwość resetowania sterty do staniu z chwili uruchomienia programu.
- Możliwość samoistnego zwiększania regionu sterty poprzez żądanie do systemu operacyjnego.
- Bezpieczeństwo wykorzystania w środowisku wielowątkowym.

Przestrzeń adresowa będzie zawsze zorganizowana jako ciąg stron **4KB**.

Funkcje do implementacji

Przedstawione poniżej funkcje należy zaimplementować zgodnie z podaną specyfikacją.

```
int heap_setup(void);
```

Funkcja `heap_setup` inicjuje (organizuje) stertę w obszarze przeznaczonej do tego pamięci. Wielkość obszaru pamięci dostępnej dla sterty nie jest znana.
W rzeczywistych przypadkach kod obsługujący stertę korzysta z funkcji systemu operacyjnego `sbrk()` o prototypie danym w pliku nagłówkowym `unisdt.h`. Na potrzeby tego projektu należy korzystać z funkcji `custom_sbrk()` o prototypie danym plikiem nagłówkowym `custom_unisdt.h`.
Projekt w którym należy przygotować swoje implementacje, oraz w którym znajduje się symulacja funkcji `sbrk` – funkcja `custom_sbrk`, dostępny jest w projekcie **`heap_sbrk-sim`** pod adresem https://github.com/tomekjaworski/SO2/tree/master/heap_sbrk-sim.

Wartość zwracana:

- 0 – jeżeli sterta została poprawnie zainicjowana
- -1 – jeżeli sterta jest uszkodzona i nie wolno z niej korzystać

```
void* heap_malloc(size_t count);
void* heap_calloc(size_t number, size_t size);
void  heap_free(void* memblock);
void* heap_realloc(void* memblock, size_t size);
```

Funkcje `heap_malloc`, `heap_calloc`, `heap_free` oraz `heap_realloc` mają zostać zaimplementowane zgodnie ze specyfikacją Biblioteki Standardowej **GNU C Library** (*glibc*), dostępnej pod adresem http://man7.org/linux/man-pages/man3/malloc.3.html. Więcej szczegółów o budowie managera sterty Biblioteki Standardowej GLIBC można znaleźć m.in. pod adresem https://sourceware.org/glibc/wiki/MallocInternals.

Zaimplementowane funkcje muszą dostarczać identycznej funkcjonalności co ich powszechnie stosowane odpowiedniki z Biblioteki Standardowej.

Wartość zwracana:

- Zależna od implementowanej funkcji

```
void* heap_malloc_debug(size_t count, int fileline, const char* filename);
void* heap_calloc_debug(size_t number, size_t size, int fileline,
                                                    const char* filename);
void* heap_realloc_debug(void* memblock, size_t size, int fileline,
                                                    const char* filename);
```

Rodzina funkcji `heap_*_debug` działa jak ich odpowiedniki bez przyrostka `_debug`, przy czym działanie to jest rozszerzone o mechanizm opisywania alokowanych/modyfikowanych bloków.

Do każdego zmodyfikowanego/zaalokowanego bloku funkcje dodają informacje o nazwie pliku źródłowego (`filename`) oraz linii (`fileline`) z której zostały wykonane. Przykładowe wywołanie takich funkcji to:

```
void* ptr = heap_malloc_debug(current_heap, 1024, __FILE__, __LINE__);
```

lub

```
#define malloc(_size) heap_malloc_debug(current_heap, (_size), __FILE__, __LINE__)
void* ptr = malloc(1024);
```

W powyższych przykładach symbole `__FILE__` oraz `__LINE__` zastępowane są przez preprocesor języka C/C++ odpowiednio nazwą pliku (np. "`test.c`" typu `const char*`) oraz numerem linii (np. 42 typu `int`)

w których te symbole zostały znalezione[1]. W przypadku makra `malloc` powyższe symbole reprezentują plik oraz linię rozwinięcia makra `malloc`.

Informacje o wierszu oraz pliku wywołania funkcji sterty znacznie ułatwiają wyszukiwanie wycieków pamięci w pisanych programach. Do tego celu służy funkcja `heap_dump_debug_information` opisana dalej w tekście, która wyświetla informacje o wszystkich zaalokowanych na stercie blokach wraz z ich rozmiarem oraz plikiem/wierszem alokacji.

```
void* heap_malloc_aligned(size_t count);
void* heap_calloc_aligned(size_t number, size_t size);
void* heap_realloc_aligned(void* memblock, size_t size);
```

Rodzina funkcji `heap_*_aligned` wykonuje operacje funkcji bazowych `*` z ograniczeniem wymuszającym alokacje nowych bloków **wyłącznie na początku strony pamięci**. W praktyce adres zwracany przez funkcje `_aligned` jest zawsze wielokrotnością długości strony pamięci i zaczyna się w jej zerowym bajcie.

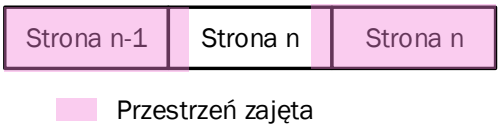
Przyjmując założenie, że:

- strona ma wielkość 4096 bajtów (0x1000) oraz
- `ptr` to zmienna przechowująca wynik alokacji pamięci funkcjami `_aligned`

następujące wyrażenie musi być zawsze prawdziwe[2]:

```
(intptr_t)ptr & (intptr_t)(PAGE_SIZE - 1) == 0
```

Przykład 2: Na stercie dostępnych jest 3000 bajtów (cześć strony *n*), zgodnie z poniższym rysunkiem:



Uruchomienie funkcji `malloc(1000)` powiedzie się i funkcja zwróci wskaźnik do nowo zaalokowanego bloku w tym obszarze. Jednak uruchomienie funkcji `malloc_aligned(1000)` nie powiedzie się (zwróci NULL). Wynika to z faktu, że w obszarze wolnym nie ma granicy stron.

Przykład 2: W przestrzeni wolnej o długości $4096n + m$ bajtów (przy $m > 0$) można zaalokować nie więcej niż n bloków na granicy stron.

---

[1] Więcej informacji na temat specjalnych symboli preprocesorami języka C/C++ można odnaleźć tutaj: https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html

[2] Typ `intptr_t` to typ całkowity będący w stanie przechować adres (istotna jest długość w bitach). Więcej informacji na temat podobnych typów danych: https://en.cppreference.com/w/c/types/integer.

```
void* heap_malloc_aligned_debug(size_t count, int fileline,
                                const char* filename);
void* heap_calloc_aligned_debug(size_t number, size_t size, int fileline,
                                const char* filename);
void* heap_realloc_aligned_debug(void* memblock, size_t size, int fileline,
                                const char* filename);
```

Funkcje `heap_*_aligned_debug` stanowią połączenie funkcji `*` z rozszerzeniem `_debug` oraz `_aligned`.

```
size_t   heap_get_used_space(void);
size_t   heap_get_largest_used_block_size(void);
uint64_t heap_get_used_blocks_count(void);
size_t   heap_get_free_space(void);
size_t   heap_get_largest_free_area(void);
uint64_t heap_get_free_gaps_count(void);
```

Funkcje te odpowiadają za zwracanie podstawowych statystyk i informacji o stercie.

- `heap_get_used_space` – Funkcja zwraca liczbę wykorzystanych bajtów sterty.
    - Do tej liczby zaliczają się zarówno zaalokowane bloki jak i struktury wewnętrzne, pozwalające na utrzymanie formatu sterty.
- `heap_get_largest_used_block_size` – Funkcja zwraca długość największego zaalokowanego bloku, lub 0 gdy niczego na niej nie zaalokowano.
- `heap_get_used_blocks_count` – Funkcja zwraca liczbę zaalokowanych bloków.
- `heap_get_free_space` – Funkcja zwraca liczbę dostępnych bajtów sterty.
    - Pamiętaj, że przestrzeń zajęta przez struktury wewnętrzne sterty nie jest przestrzenią dostępną do zaalokowania.
- `heap_get_largest_free_area` – Funkcja zwraca długość największego wolnego obszaru dostępnego na stercie.
    - W przypadku pustej sterty (brak zaalokowanych bloków) wartość zwracana przez tę funkcję powinna być równa wartości funkcji `heap_get_free_space`.
- `heap_get_free_gaps_count` – Funkcja zwraca liczbę wolnych obszarów, w których można zaalokować blok o długości nie mniejszej niż długość słowa danych CPU.
    - Dla przykładu, jeżeli program jest skompilowany dla x86-64 to długość słowa wynosi 64 bity (8 bajtów[3]). Ponadto przykładowy alokator dołącza do każdego alokowanego bloku 64 bajty na strukturę opisującą ten blok i jego sąsiedztwo. Oznacza to, że najmniejsza przestrzeń międzyblokowa, aby była zauważona prze funkcję `heap_get_free_gaps_count`, musi mieć co najmniej 8 + 64 bajty wielkości.

[3] Patrz `sizeof(void*)`.

```
enum pointer_type_t get_pointer_type(const const void* pointer);
```

Funkcja `get_pointer_type` zwraca informację o przynależności wskaźnika `pointer` do sterty lub pamięci spoza sterty. Funkcja ta, na podstawie informacji zawartych w strukturze sterty klasyfikuje wskaźnik `pointer` i zwraca jedną z wartości typu `pointer_type_t`.

```
enum pointer_type_t
{
	pointer_null,
	pointer_out_of_heap,
	pointer_control_block,
	pointer_inside_data_block,
	pointer_unallocated,
	pointer_valid
};
```

Wartości typu `pointer_type_t` mają następującą interpretację:

- `pointer_null` – przekazany wskaźnik jest pusty – posiada wartość `NULL`.
- `pointer_out_of_heap` – przekazany wskaźnik nie leży w obszarze sterty.
- `pointer_control_block` – przekazany wskaźnik leży w obszarze sterty, ale wskazuje na obszar struktur wewnętrznych.
- `pointer_inside_data_block` – przekazany wskaźnik wskazuje środek jakiegoś zaalokowanego bloku. Przez środek należy rozumieć adres bajta innego niż pierwszego danego bloku.
- `pointer_unallocated` – przekazany wskaźnik wskazuje na obszar wolny (niezaalokowany).
- `pointer_valid` – przekazany wskaźnik jest poprawny – wskazuje na pierwszy bajt bloku.
    - Każdy wskaźnik zwracany przez `malloc/calloc/realloc` musi być typu `pointer_valid`. I tylko takie wskaźniki ma przyjmować funkcja `free`.

```
void* heap_get_data_block_start(const void* pointer);
```

Funkcja zwraca wskaźnik na początek bloku, na którego dowolny bajt wskazuje `pointer`.

Wartość zwracana:

- Jeżeli wskaźnik `pointer` nie wskazuje na dowolny bajt dowolnego bloku (czyli nie jest typu `pointer_inside_data_block` oraz `pointer_valid`) to funkcja `heap_get_data_block_start` zwraca `NULL`.
- Ponadto, jeżeli `pointer` jest typu `pointer_valid` to funkcja `heap_get_data_block_start` zwraca pointer (bez zmian).

```
size_t heap_get_block_size(const const void* memblock);
```

Funkcja zwraca długość bloku danego wskaźnikiem `memblock`. Jeżeli `memblock` nie wskazuje na zaalokowany blok (inny niż `pointer_valid`), to funkcja zwraca 0.

```
int heap_validate(void);
```

Funkcja wykonuje sprawdzenie spójności sterty.

Wartość zwracana:

- 0 – jeżeli sterta jest poprawna
- -1 – jeżeli sterta jest uszkodzona

Uwaga! Wbrew pozorom jest to funkcja najtrudniejsza do napisania. Musi ona być odporna **na wszelkie możliwe** uszkodzenia sterty – nie ma prawa doprowadzić do przerwania działania programu ze względu na błąd dostępu do pamięci.

Podstawowym pytaniem, jakie musi zadać sobie projektant takiej funkcji jest następujące: *W jaki sposób sprawdzić, czy struktura opisująca stertę nie została uszkodzona?* Ponieważ jeżeli nie została uszkodzona, to można z niej odczytać granice regionu przydzielonego stercie (patrz funkcja `heap_setup`) i względem tej granicy walidować każdy wskaźnik w strukturach wewnętrznych sterty.

Dla ułatwienia można zwiększyć liczbę typów błędów raportowanych przez `heap_validate` ponad podane -1 (np. -2 – brak terminatorów, -3 – uszkodzone płotki, -4 – błędy sum kontrolnych, itd..).

```
void heap_dump_debug_information(void);
```

Funkcja wyświetla informacje o blokach zaalokowanych na stercie, wraz z podsumowaniem. Informacje te powinny zawierać:

- Listę wszystkich zaalokowanych bloków ze wskazaniem:
  - Adresu bloku,
  - długości bloku w bajtach,
  - nazwy pliku źródłowego, w którym nastąpiła alokacja (jeżeli użyto funkcji _debug),
  - numeru linii pliku źródłowego, w której nastąpiła alokacja (j/w).
- wielkość sterty w bajtach,
- liczbę bajtów zajętych,
- liczbę bajtów do zaalokowania,
- wielkość największego wolnego bloku.

Uwagi dodatkowe

- Do każdej funkcji należy przygotować zestaw testów jednostkowych, pozwalających na kontrolowane uzyskanie wszystkich możliwych wartości zwracanych (np. dla `malloc` będzie to `NULL` oraz nie-`NULL`) – należy przetestować zarówno scenariusze pozytywne jak i negatywne/awarie.
- Każdy test jednostkowy musi zostać wyposażony w krótki opis słowny, uzasadniający jego istnienie.
- W programie nie wolno używać wszelkich mechanizmów alokacji pamięci. Za wyjątkiem swoich 😊

Przykładowy test

Testowa funkcja `main` dostępna jest w oficjalnym repozytorium przedmiotu: https://github.com/tomekjaworski/SO2/blob/54b471147bea079d4885fb7947dab5e0f1189966/heap_sbrk-sim/main.c.