Systemy operacyjne (zaawansowane)

Projekt nr 1

Termin oddawania 7 stycznia 2018

Zaimplementuj w języku C odpowiednik bibliotecznego algorytmu zarządzania pamięcią. Należy udostępnić procedury o poniższych sygnaturach – ich znaczenie jest dokładnie opisane w podręcznikach systemowych `malloc(3)` i `posix_memalign(3)`.

```
void *malloc(size_t size);
void *calloc(size_t count, size_t size);
void *realloc(void *ptr, size_t size);
int   posix_memalign(void **memptr, size_t alignment, size_t size);
void  free(void *ptr);
```

Projekt musi kompilować się bez błędów i ostrzeżeń (opcje `-std=gnu11 -Wall -Wextra`) kompilatorem `gcc` lub `clang` pod systemem `Linux`. Należy dostarczyć plik `Makefile`, tak by po wywołaniu polecenia «`make`» otrzymać pliki binarne, a po «`make clean`» pozostawić w katalogu tylko pliki źródłowe. Rozwiązanie należy zamieścić w systemie oddawania zadań na stronie zajęć.

Przygotowanie

Zanim przystąpisz do programowania dokładnie przemyśl i rozpisz na kartce organizację struktur danych w pamięci oraz działanie poszczególnych operacji – zaoszczędzisz sobie w ten sposób czasu na odpluskwianie, które nie będzie zbyt przyjemne. Mimo wszystko zapewne będziecie zmuszeni do użycia `GNU debugger`. Rozpoczęcie znajomości z tym bardzo potężnym, lecz niezbyt przyjaznym, narzędziem należy zacząć od przeczytania samouczka Richard Stallman's gdb Debugger Tutorial[1].

Można samemu zaprogramować niezbędne struktury danych, ale dopuszczalne jest użycie gotowych implementacji z systemów `BSD`. Jeśli potrzebujesz list dwukierunkowych to są one dostępne w pliku nagłówkowym queue.h[2] i opisane w podręczniku `queue(3)`[3]. Ponieważ implementacja list bazuje na makrach preprocesora pamiętaj, aby nie przekazywać do parametrów makr wyrażeń mających efekty uboczne. Dopuszczalne jest również użycie implementacji bitmap (bitstring.h[4]) oraz drzew rozchylanych i drzew czerwono-czarnych (tree.h[5]) opisanych odpowiednio w `bitstring(3)`[6] i `tree(3)`[7].

Nie zapomnij skrupulatnie przeczytać specyfikacji procedur bibliotecznego menadżera pamięci! Lepiej dobrze zrozumieć co się implementuje przed rozpoczęciem prac programistycznych.

[1] `http://web.archive.org/web/20161117202458/http://www.unknownroad.com/rtfm/gdbtut/`
[2] `http://bxr.su/NetBSD/sys/sys/queue.h`
[3] `https://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3`
[4] `http://bxr.su/FreeBSD/sys/sys/bitstring.h`
[5] `http://bxr.su/NetBSD/sys/sys/tree.h`
[6] `https://www.freebsd.org/cgi/man.cgi?query=bitstring&sektion=3`
[7] `https://www.freebsd.org/cgi/man.cgi?query=tree&sektion=3`

Specyfikacja

Niech **obszar** będzie przedziałem adresów wirtualnych, pod które podczepiono strony z użyciem wywołania systemowego `mmap(2)` z flagą `MAP_ANONYMOUS`. Rozmiar strony można pobrać procedurą `getpagesize(2)`. Obszar należy zwrócić do systemu z użyciem `munmap(2)`, jeśli zawiera wyłącznie jeden wolny **blok**, a ilość wolnego miejsca w pozostałych obszarach przekracza ustalony próg (np. kilka stron). Pamięcią dostępną w obrębie obszaru należy zarządzać algorytmem przydziału bloków, który opisano poniżej. Jeśli użytkownik zażądał bloku o rozmiarze większym niż kilka stron, to należy poświęcić na to cały jeden obszar. Poniżej podano propozycję rekordu obszaru:

```
1 typedef struct mem_chunk {
2   LIST_ENTRY(mem_chunk) ma_node;      /* node on list of all chunks */
3   LIST_HEAD(, mem_block) ma_freeblks; /* list of all free blocks in the chunk */
4   int32_t size;                       /* chunk size minus sizeof(mem_chunk_t) */
5   mem_block_t ma_first;               /* first block in the chunk */
6 } mem_chunk_t;
7
8 LIST_HEAD(, mem_chunk) chunk_list; /* list of all chunks */
```

Zarządzanie blokami ma bazować na liście dwukierunkowej posortowanej względem adresów. Do przydziału bloku używaj strategii **first-fit**. Przy zwalnianiu bloków gorliwie wykonuj operację scalania, aby zrobić to szybko wykorzystaj technikę „boundary tag". Pamiętaj, że adresy zwracane przez procedurę `malloc` muszą być podzielne przez rozmiar największego słowa maszynowego. Poniżej podano propozycję rekordu bloku:

```
1 typedef struct mem_block {
2   int32_t mb_size;                 /* mb_size > 0 => free, mb_size < 0 => allocated */
3   union {
4     LIST_ENTRY(mem_block) mb_node; /* node on free block list, valid if block is free */
5     uint64_t mb_data[0];           /* user data pointer, valid if block is allocated */
6   };
7 } mem_block_t;
```

Algorytm zachowywać się poprawnie w programach wielowątkowych – należy zadbać o zakładanie blokad `pthread_mutex_lock(3)` w odpowiednich miejscach. Udostępnij procedurę «`mdump`» drukującą stan menadżera pamięci – tj. listy wszystkich obszarów oraz bloków. Na pewno będzie przydatna do odpluskwiania. Pamiętaj, że operacja «`realloc`» przy zmniejszaniu długości bloku nie powinna przenosić go w inne miejsce. Natomiast przy zwiększaniu długości bloku nie powinna go przenosić, jeśli jest za nim wystarczająca ilość wolnego miejsca na jego powiększenie.

Testowanie

Na początku napisz testy jednostkowe z użyciem biblioteki minunit[8]. Do swojego katalogu projektu skopiuj plik «`minuint.h`». Sposób konstruowania testów podano w pliku «`minunit_example.h`». By uniknąć kolizji symboli z procedurami biblioteki standardowej, wszystkim procedurom własnego algorytmu zarządzania pamięcią należy dodać prefiks (np. `foo_`).

Następnie przystosuj swój kod do przesłonięcia bibliotecznego menadżera pamięci. W tym celu wygeneruj aliasy procedur (`foo_malloc` → `malloc`) i skonsoliduj swoje rozwiązanie jako bibliotekę współdzieloną. Nadając odpowiednią wartość zmiennej środowiskowej `LD_PRELOAD` opisanej w podręczniku `ld.so(8)` przesłoń symbole z biblioteki standardowej. Zmusisz w ten sposób wybrane programy do korzystania z własnego algorytmu. Przed doborem programów testowych upewnij się poleceniem `ltrace(1)`, że korzystają one z procedur bibliotecznego algorytmu zarządzania pamięcią.

[8] `https://github.com/siu/minunit`

Ocena

Implementacja zadania opisana do tej pory jest warta 6 punktów. Pod uwagę będą brane:

- przejrzystość rozwiązania – należy używać prostych funkcji otwartych (ang. *inline*),
- jakość testów jednostkowych – należy przemyśleć strategię testowania,
- przystosowanie implementacji do odpluskwiania – np. zrzucanie dziennika operacji na standardowe wyjście błędów «`stderr`» przy ustawieniu zmiennej środowiskowej «`MALLOC_DEBUG`»,
- czytelność zrzutu stanu menadżera pamięci,
- poprawność zachowania wybranych programów testowych.

Punkty dodatkowe będą przyznawane za:

- wykrywanie uszkodzeń struktur danych menadżera pamięci,
- identyfikację niewłaściwych adresów przekazywanych do «`free`»,
- szybsze znajdowanie wolnych bloków ustalonego rozmiaru,
- zmniejszenie objętości struktur danych do przechowywania informacji o małych blokach,
- ograniczenie współzawodnictwa między wątkami używającymi menadżera pamięci.

Liczba przyznawanych punktów bonusowych jest nieustalona, a będzie zależeć od stopnia zaawansowania i przetestowania poszczególnych usprawnień algorytmu.