Systemy operacyjne

Projekt programistyczny nr 2

„Menadżer pamięci"

Termin oddawania: 26 stycznia 2020

Wprowadzenie

Twoim zadaniem jest zaimplementowanie algorytmu dynamicznego zarządzania pamięcią dla programów napisanych w języku C. Wykonanie zadania będzie wymagało podjęcia wielu decyzji, które będą przekładać się na wydajność operacji «malloc» oraz «free», a także oszczędność zużycia pamięci, tj. fragmentację wewnętrzną i zewnętrzną.

Wskazówka: Przed przystąpieniem do pracy należy zapoznać się z rozdziałem §9.9 książki CSAPP.

Wymogi formalne

Projekt musi kompilować się bez błędów i ostrzeżeń kompilatorem gcc 8.3 lub clang 7.0 pod systemem Debian GNU/Linux 10 dla architektury x86-64. Należy wykorzystać zamieszczony na stronie przedmiotu szkielet rozwiązania. Zawiera on również program «mdriver» testujący poprawność oraz wydajność dostarczonego algorytmu zarządzania pamięcią. Program ten będzie używany do oceny rozwiązania studenta.

Po upewnieniu się, że dostarczony szkielet rozwiązania kompiluje się i działa, należy zastąpić zawartość pliku «mm.c» własną implementacją. W komentarzu w nagłówku pliku «mm.c» należy wpisać swoje imię, nazwisko oraz numer indeksu, oświadczając, że jest się jedynym autorem kodu źródłowego.

Możesz wykorzystać fragmenty kodu dostępne w książce CSAPP lub pliku «mm-implicit.c». Dozwolona jest również adaptacja kodu źródłowego[1] drzew rozchylanych (ang. *splay tree*) autorstwa Daniela Sleatora. W takim przypadku należy jasno zaznaczyć w kodzie źródłowym które fragmenty nie są Twojego autorstwa. Jeśli zainspirowały Ciebie jakieś publikacje lub książki, to należy podać do nich odnośniki.

Nie można korzystać z żadnych innych źródeł bez zgody wykładowcy. Haniebnym jest by skopiować lub zmodyfikować rozwiązanie studentów z naszej lub innych uczelni, lub wykorzystać inne implementacje dostępne w zasobach sieci Internet, i przedstawić jako własne rozwiązanie.

Po wykonaniu zadania należy umieścić w systemie SKOS wyłącznie plik «mm.c».

UWAGA! Wykrycie niesamodzielnej pracy będzie skutkować natychmiastowym wydaleniem studenta z kursu, zgłoszeniem przypadku do komisji antyplagiatowej i rozmową z dziekanem dotyczącą kodeksu studiowania.

Specyfikacja

W pliku «mm.c» dostarczono najprostszą poprawną implementację algorytmu zarządzania pamięcią. Jej głównym mankamentem jest brak zwalniania przydzielonych bloków. Zapoznaj się z implementacją poszczególnych procedur, żeby lepiej zrozumieć zadania menadżera pamięci.

Twój algorytm zarządzania pamięcią będzie składał się z następujących procedur:

- «int mm_init(void)»: Przed pierwszym wywołaniem «mm_malloc», «mm_realloc» lub «mm_free», program «mdriver» zawoła «mm_init», żeby wykonać inicjację algorytmu zarządzania pamięcią. W trakcie testów procedura ta będzie wołana wielokrotnie. Jest to właściwe miejsce na ustawienie

[1] http://www.link.cs.cmu.edu/link/ftp-site/splaying/top-down-splay.c

wszystkich zmiennych globalnych, z których będziesz korzystać. Procedura ma zwrócić -1, jeśli inicjacja się nie powiodła, w przeciwnym wypadku 0.

- «`void *mm_malloc(size_t size)`»: Zwraca blok pamięci zawierający co najmniej «`size`» bajtów dostępnych dla użytkownika. Blok musi leżeć w obrębie sterty zarządzanej przez algorytm.

  Twoje rozwiązanie będzie porównywane z implementacją algorytmu z biblioteki standardowej języka `C`. Zatem adres zwracanego bloku musi być podzielny przez 16 (stała «`ALIGNMENT`»).
- «`void mm_free(void *ptr)`»: Zwalnia blok o adresie zwróconym przez «`mm_malloc`» lub «`mm_realloc`». Nie musisz sprawdzać, czy przekazany wskaźnik jest poprawny. Procedura nie robi nic, jeśli przekazany argument jest równy «`NULL`».
- «`void *mm_realloc(void *ptr, size_t size)`»: Podobnie jak «`mm_malloc`» zwraca wskaźnik do przydzielonego bloku, ale ma kilka scenariuszy użycia:
  - jeśli «`ptr`» jest równy «`NULL`», to wywołanie jest tożsame z «`mm_malloc(size)`»,
  - jeśli «`size`» jest równy 0, to wywołanie jest tożsame z «`mm_free(ptr)`»,
  - jeśli «`ptr`» nie jest równy «`NULL`», to musiał zostać zwrócony przez procedurę «`mm_malloc`» lub «`mm_realloc`», i należy zmienić rozmiar przydzielonego bloku. Dopuszcza się przeniesienie bloku pod nowy adres różny od «`ptr`». Jeśli użytkownik zwiększa rozmiar bloku, to dodatkowa pamięć w bloku musi pozostać niezainicjowana.
- «`void *mm_check(int verbose)`»: Sprawdza poprawność danych algorytmu zarządzania pamięcią.

Semantyka powyższych procedur zgadza się z przedstawioną w podręczniku systemowym `malloc(3)`.

Procedury zarządzania stertą

Plik «`memlib.c`» zawiera procedury, których zadaniem jest symulacja podsystemu zarządzania pamięcią udostępnianego przez jądro SO. Należy je wykorzystać do implementacji menadżera pamięci.

- «`void *mem_sbrk(long incr)`»: Rozszerza stertę o dokładnie «`incr`» bajtów i zwraca adres pierwszego bajtu nowo przydzielonego obszaru sterty. Zachowuje się dokładnie tak samo jak `sbrk(2)`, za wyjątkiem tego, że nie przyjmuje ujemnych argumentów, tj. nie można zmniejszać rozmiaru sterty.
- «`void *mem_heap_lo(void)`»: Zwraca adres pierwszego bajtu należącego do sterty.
- «`void *mem_heap_hi(void)`»: Zwraca adres ostatniego bajtu należącego do sterty.
- «`size_t mem_heapsize(void)`»: Zwraca rozmiar sterty w bajtach.
- «`size_t mem_pagesize(void)`»: Zwraca rozmiar strony pamięci wirtualnej (zazwyczaj 4KiB).

UWAGA! Twój algorytm może używać tylko i wyłącznie pamięci przydzielonej w obrębie sterty! Do przechowywania stanu menadżera pamięci możesz używać niezainicjowanych zmiennych globalnych o typach skalarnych. Typy tablicowe są niedozwolone!

Sprawdzanie poprawności

Nawet najbardziej doświadczeni programiści mogą mieć sporo problemów z implementacją algorytmu zarządzania pamięcią. Głównymi winowajcami są: arytmetyka na wskaźnikach i nietypowane wskaźniki, których niestety nie da się uniknąć. Dlatego kluczowym jest, by zaimplementować procedurę sprawdzającą spójność danych menadżera pamięci.

Poniżej podano kilka przykładów niezmienników, które powinny być zachowane między wywołaniami procedur algorytmu zarządzania pamięcią:

- Każdy blok na liście wolnych bloków jest oznaczony jako wolny.
- Każdy blok oznaczony jako wolny jest na liście wszystkich wolnych bloków.
- Nie istnieją dwa przyległe do siebie bloki, które są wolne.
- Wskaźnik na poprzedni i następny blok odnoszą się do adresów należących do zarządzanej sterty.
- Wskaźniki na liście wolnych bloków wskazują na początki wolnych bloków.

Procedura «mm_check» powinna zwrócić 0, jeśli wykryje naruszenie jednego z niezmienników. Jeśli parametr «verbose» jest niezerowy to możesz wydrukować dane menadżera pamięci, np.: listę wszystkich bloków i zawartość listy wolnych bloków. Dzięki temu lepiej zrozumiesz przyczyny występowania błędu. O ile to możliwe warto wydrukować zawartość sterty również w przypadku, gdy jakiś niezmiennik został naruszony.

UWAGA! Pamiętaj, żeby przed zamieszczeniem rozwiązania w systemie SKOS usunąć drukowanie zbędnych komunikatów diagnostycznych. W przeciwnym wypadku pomiar wydajności algorytmu zostanie zaburzony na Twoją niekorzyść.

Program `mdriver`

Program «mdriver» służy do badania poprawności, przepustowości oraz stopnia wykorzystania sterty dla algorytmów zarządzania pamięcią. W tym celu odtwarza sekwencje operacji «malloc», «realloc» i «free», które utworzono na podstawie wykonania różnych programów. Ślady są dostępne w katalogu «traces». Program «mdriver» przyjmuje następujące parametry z linii poleceń:

- «-f <file>»: użyj pliku <file> zamiast domyślnego zestawu śladów,
- «-c <file>»: j.w. ale sprawdź wyłącznie poprawność,
- «-l»: Najpierw uruchom wszystkie testy na implementacji menadżera pamięci z biblioteki standardowej języka C, a potem na rozwiązaniu studenta.
- «-V»: Drukuje komunikaty diagnostyczne związane z przetwarzaniem poszczególnych śladów.
- «-D»: Sprawdza poprawność danych algorytmu zarządzania pamięcią, przy pomocy procedury «mm_check», po wywołaniu wykonaniu każdej operacji.

Wskazówki od wykładowcy

- Przemyśl sobie dokładnie algorytm zanim przystąpisz do programowania. Rozpisz na kartce organizację sterty, strukturę bloku i działanie poszczególnych operacji. W ten sposób masz szansę zaoszczędzić sporo czasu na odpluskwianie, które będzie czasochłonne i niezbyt przyjemne.
- Podziel implementację na etapy. Skoncentruj się na tym, by jak najszybciej osiągnąć działający krok pośredni. Odpluskwianie jest cięższe, kiedy masz mnóstwo nieprzetestowanego kodu. Lepiej jest dodawać optymalizacje do poprawnego kodu.
- Zacznij od uruchamiania «mdriver -f» na krótkich śladach. W początkowej fazie implementacji pozwoli Ci to wyłapać większość prostych błędów. Szczególnie przydatne ślady to «short-*.rep».
- Zmień opcję kompilacji «-O3» na «-Og» na czas odpluskwiania. Dzięki analizie stanu programu przy użyciu gdb zrozumiesz co doprowadziło do awarii. Przyda się polecenie «backtrace».
- Wołaj procedurę «mm_check» w trakcie odpluskwiania. Pozwoli Ci to zawęzić obszar poszukiwania usterki psującej jeden z niezmienników. Może chcesz sprawdzać niezmienniki częściej niż «mdriver»?
- Używaj funkcji pomocniczych zdefiniowanych z użyciem «static inline». Kompilator efektywnie je optymalizuje. Arytmetykę na wskaźnikach lepiej zawrzeć w krótkiej funkcji niż powtarzać wielokrotnie ten sam kod i niepotrzebnie wystawić się na błąd typu „kopiuj-wklej". Kod będzie bardziej przejrzysty, więc i prościej będzie w nim znaleźć usterki.
- Zrób sobie przerwę, jeśli szukasz błędu przez dłuższy czas. Dokładnie przeczytaj swój kod. Upewnij się, że rozumiesz każdy jego kawałek. Może coś Ci umknęło? (Może warto przewietrzyć mózg i wyprowadzić psa?)
- Nie zwlekaj z rozpoczęciem pracy. Dobry algorytm zarządzania pamięcią można zawrzeć w 400 liniach kodu. Dla sporej części z Was będzie to jednak spore wyzwanie. Warto dać sobie trochę czasu na przemyślenie problemu. Próba ukończenia zadania na ostatnią chwilę raczej zakończy się porażką.

Prowadzący zajęcia zaprogramował kilka wariantów zadania. Szkielet pierwszego z nich zamieścił w pliku «mm-implicit.c» dla studentów, którzy chcieliby podążać podobną ścieżką.

Na początku warto poczynić kilka ważnych obserwacji. Rozmiar sterty jest ograniczony stałą «MAX_HEAP». Możemy założyć, że jest mniejszy niż 4GiB. Sterta zaczyna się pod adresem, który nie mieści się w 32-bitowym słowie. Rozmiar bloku jest wielokrotnością 16 bajtów. Jest to o tyle istotne, że musimy zagospodarować pozostałe miejsce w bloku na przechowywanie węzła struktury danych służącej do szybkiego wyszukiwania

wolnych bloków. Jeśli przyjmiemy, że minimalny rozmiar bloku wynosi 16 bajtów, to ile miejsca zostanie nam na węzeł listy dwukierunkowej lub drzewa zbalansowanego? Czy skompresujemy wskaźniki do 4 bajtów?

Wydaje się, że najlepszą strategią rozwiązywania tego zadania jest podzielenie go na następujące duże kroki:

1. Implementacja najprostszego algorytmu bazującego na boundary tags. Niech «`mm_malloc`» wykorzystuje politykę first-fit, a «`mm_free`» gorliwie złącza wolne bloki.
2. Użycie optymalizacji boundary tags opisanej na wykładzie. Należy zmaksymalizować przestrzeń dostępną dla użytkownika. Następnie można zoptymalizować «`mm_realloc`».
3. Użycie listy dwukierunkowej do wyszukiwania wolnych bloków. Porównanie polityki wstawiania: LIFO, FIFO i posortowanie względem adresu bloku. Porównanie polityk first-fit, best-fit i next-fit.
4. Optymalizacja wyszukiwania wolnych bloków z użyciem nieco bardziej zaawansowanych struktur danych np.: segregated fits albo splay tree. Może lista ostatnio używanych bloków.
5. Dodatkowe optymalizacje. Podział sterty na 64KiB bloki i kompresja wskaźników do 2 bajtów. Używanie bitmapy do zarządzania małymi blokami. (Generalnie co komu w duszy zagra, jeśli będzie mieć czas.)

Ocena

Prowadzący będą używać automatycznego systemu oceny zadań, zatem musisz uważać, by dostarczone rozwiązanie było poprawne. Możesz nie otrzymać punktów, jeśli program nie skompiluje się, zakończy błędem, zawiesi się, drukuje na ekran zbędne komunikaty powodując znaczne spowolnienie odtwarzania śladów.

Twoje rozwiązanie będzie testowane na jawnych śladach, dostarczonych ze szkieletem rozwiązania, jak i niejawnych śladach wygenerowanych wyłącznie na użytek prowadzących ćwiczenio-pracownię. Program «`mdriver`» przydzieli najgorszą możliwą ocenę za ślad, na którym algorytm zawiedzie.

Twoja implementacja musi obsługiwać wszelkie poprawne ślady operacji. Niemniej możesz optymalizować swój algorytm pod kątem dostarczonych śladów tak, by uzyskać jak najwyższą ocenę. W szczególności możesz przeanalizować ślady, dla których Twój algorytm sprawuje się najgorzej, a następnie zaprojektować optymalizacje ogólne albo wykorzystujące jakąś wiedzę o dostarczonych śladach.

Implementacja menadżera pamięci jest warta maksymalnie 20 punktów. Na ostateczną ocenę będą miały wpływ następujące czynniki:

- (40%) Efektywność wykorzystania sterty. Dla każdej operacji «`malloc`» i «`realloc`» program «`mdriver`» mierzy stosunek liczby wszystkich bajtów przydzielonych użytkownikowi do rozmiaru sterty. Ponieważ nie można zmniejszyć rozmiaru sterty wystarczy, że program wyznaczy maksymalną liczbę bajtów przydzieloną użytkownikowi i podzieli ją przez końcowy rozmiar sterty.
- (40%) Przepustowość. Program «`mdriver`» wyznacza uśrednioną liczbę operacji na sekundę. Żeby otrzymać maksymalną liczbę punktów Twoja implementacja musi być mniej więcej tak samo szybka jak biblioteczny algorytm zarządzania pamięcią.
- (15%) Czytelność kodu.
  - Kod ma być podzielony na procedury o zwięzłych nazwach dobrze tłumaczących ich przeznaczenie.
  - Przed każdą większą procedurą należy umieścić komentarz, który opisuje jej zadanie i tłumaczy bardziej zawiłe detale jej działania.
  - Plik źródłowy musi zaczynać komentarzem, w którym zawarto: szczegółowy opis zawartości zajętego i wolnego bloku pamięci, wysokopoziomowy opis działania przydziału i zwalniania bloku.
  - Procedura sprawdzająca spójność struktur danych algorytmu powinna wykrywać jak najwięcej usterek i być dobrze udokumentowana.
- (5%) Poprawność działania na niejawnych śladach.