

Systemy operacyjne

Lista zadań nr 8

Na zajęcia 3 grudnia 2020

Należy przygotować się do zajęć czytając następujące rozdziały książek lub publikacji:

- **Dynamic Storage Allocation: A Survey and Critical Review**¹: 1 – 3

UWAGA! W trakcie prezentacji należy być gotowym do zdefiniowania pojęć oznaczonych **wytłuszczoną** czcionką.

Zadanie 1. Systemy uniksowe udostępniają wywołania systemowe **sbrk(2)** oraz parę **mmap(2)** i **munmap(2)**. Służą one do przydziału stron na użytek bibliotecznego algorytmu zarządzania pamięcią. Czemu implementacje **malloc(3)** preferują drugą opcję? Wyjaśnij to odwołując się do mapy pamięci wirtualnej procesu.

Wskazówka: Rozważ scenariusz, w którym proces zwolnił dużo pamięci przydzielonej na początku jego życia.

Zadanie 2. Wyjaśnij różnicę między **fragmentacją wewnętrzną** i **zewnętrzną**. Czemu nie można zastosować **kompaktowania** w bibliotecznym algorytmie przydziału pamięci? Na podstawie §2.3 opowiedz o dwóch głównych przyczynach występowania fragmentacji zewnętrznej.

Zadanie 3. Posługując się wykresem wykorzystania pamięci w trakcie życia procesu opowiedz o trzech wzorcach przydziału pamięci występujących w programach (§2.4). Na podstawie paragrafu zatytułowanego „Exploiting ordering and size dependencies” wyjaśnij jaki jest związek między czasem życia bloku, a jego rozmiarem? Wyjaśnij różnice między politykami znajdowania wolnych bloków: **first-fit**, **next-fit** i **best-fit**. Na podstawie §3.4 wymień ich słabe i mocne strony.

Zadanie 4. Algorytm przydziału pamięci udostępnia funkcje o sygnaturach «alloc: words -> @id» i «free: @id -> void» i ma do dyspozycji obszar 50 słów maszynowych. Implementacja używa dwukierunkowej listy wolnych bloków oraz boundary tags. Wyszukiwanie wolnych bloków działa zgodnie z polityką best-fit. Operacja zwalniania **gorliwie łączy** bloki i wstawia wolne bloki na początek listy. Posługując się diagramem z wykładu wykonaj krokową symulację algorytmu przydziału pamięci dla poniższego ciągu żądań. Należy wziąć pod uwagę miejsce zajmowane przez struktury danych algorytmu przydziału oraz nieużytki.

```
alloc(5) alloc(12) alloc(15) alloc(8) free(@2) free(@1) free(@3) alloc(10)
```

Uwaga: Funkcja «alloc» zwraca bloki o kolejnych identyfikatorach począwszy od @1. Adresy są wyrównane do długości słowa.

Zadanie 5. Rozważmy **algorytm kubełkowy** (§3.6) (ang. *segregated-fit*) przydziału pamięci z gorliwym łączeniem wolnych bloków. Jak przebiegają operacje «malloc» i «free»? Co robi «malloc», gdy na danej liście nie ma wolnego bloku żadanego rozmiaru? Jak poradzić sobie w trakcie łączenia wolnych bloków w procedurze «free», jeśli chcemy usunąć ostatni element z listy? Rozważ zastosowanie **leniwego łączenia** wolnych bloków w algorytmie kubełkowym przydziału pamięci – jakie problemy zauważasz?

¹<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.275>

Ściągnij ze strony przedmiotu archiwum «so20_lista_8.tar.gz», następnie rozpakuj i zapoznaj się z dostarczonymi plikami.

UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczone w komentarzu napisem «TODO».

UWAGA! Dla metod przydziału pamięci użytych w poniższych zadaniach należy być przygotowanym na wyjaśnienie:

- jak wygląda struktura danych przechowująca informację o zajętych i wolnych blokach?
- jak przebiegają operacje «alloc» i «free»?
- jaka jest pesymistyczna złożoność czasowa powyższych operacji?
- jaki jest narzut (ang. *overhead*) pamięciowy **metadanych** (tj. ile bitów lub na jeden blok)?
- jaki jest maksymalny rozmiar **nieużytków** (ang. *waste*)?
- czy w danym przypadku **fragmentacja wewnętrzna** lub **zewnętrzna** jest istotnym problemem?

Zadanie 6. Program «objpool» zawiera implementację **bitmapowego przydziału** bloków o stałym rozmiarze. Algorytm zarządza pamięcią w obrębie aren przechowywanych na jednokierunkowej liście «arenas». Pamięć dla aren jest pobierana od systemu z użyciem wywołania **mmap(2)**. Areny posiadają nagłówek przechowujący węzeł listy i dodatkowe dane algorytmu przydziału. Za nagłówkiem areny znajduje się pamięć na metadane, a także bloki pamięci przydzielane i zwalniane funkcjami «alloc_block» i «free_block».

Używając funkcji opisanych w **bitstring(3)** uzupełnij brakujące fragmenty procedur. Metadane w postaci bitmapy są przechowywane za końcem nagłówka areny. Ponieważ odpluskwanie algorytmu może być ciężkie, należy korzystać z funkcji **assert(3)** do wymuszania warunków wstępnych procedur. Twoja implementacja algorytmu zarządzania pamięcią musi przechodzić test wbudowany w skompilowany program «objpool».

Zadanie 7 (bonus). Zoptymalizuj procedurę «alloc_block» z poprzedniego zadania. Główną przyczyną niskiej wydajności jest użycie funkcji «bit_ffc». Należy wykorzystać dwa sposoby na jej przyspieszenie (a) użycie jednocyklowej instrukcji procesora **ffs²** wyznaczającej numer pierwszego ustawionego bitu w słowie maszynowym (b) użycie wielopoziomowej struktury bitmapy, tj. wyzerowany i -ty bit w bitmapie poziomu k mówi, że w i -tym słowie maszynowym bitmapy poziomu $k + 1$ występuje co najmniej jeden wyzerowany bit.

Komentarz: Bardzo dobry algorytm musiałby jeszcze wziąć pod uwagę strukturę pamięci podręcznej procesora.

Zadanie 8 (2). (Implementację zadania dostarczył Piotr Polesiuk.)

Program «stralloc» implementuje algorytm zarządzania pamięcią wyspecjalizowany pod kątem przydziału miejsca dla ciągów znakowych nie dłuższych niż «MAX_LENGTH». Ponieważ algorytm wie, że w blokach będą składowane ciągi znakowe, to nie musi dbać o wyrównanie adresu zwracanego przez procedurę «stralloc».

Podobnie jak w programie «objpool» będziemy zarządzać pamięcią dostępną za nagłówkiem areny. W obszarze tym zakodujemy **niejawną listę** (ang. *implicit list*) jednokierunkową, której węzły są kodowane w pierwszym bajcie bloku. Wartość bezwzględna nagłówka bloku wyznacza jego długość, a znak dodatni i ujemny kodują to czy blok jest wolny, czy zajęty. Nagłówek bloku o wartości zero koduje koniec listy. Ponieważ domyślnie arena ma długość 65536 bajtów to procedura «init_chunk» musi wypełnić zarządzany obszar wolnymi blokami nie większymi niż «MAX_LENGTH+1».

Twoim zadaniem jest uzupełnienie brakujących fragmentów procedur «alloc_block» i «strfree». Pierwsza z nich jest zdecydowanie trudniejsza i wymaga obsłużenia aż pięciu przypadków. Będzie trzeba dzielić bloki (ang. *splitting*), łączyć (ang. *coalescing*) lub zmieniać rozmiar dwóch występujących po sobie wolnych bloków, jeśli nie da się ich łączyć. Druga procedura jest dużo prostsza i zaledwie zmienia stan bloku upewniwszy się wcześniej, że użytkownik podał prawidłowy wskaźnik na blok.

Przed przystąpieniem do rozwiązywania przemyśl dokładnie działanie procedur. Pomyłki będą ciężkie do znalezienia. Jedyną linią obrony będzie tutaj obfite sprawdzanie warunków wstępnych funkcją **assert(3)**.

Rozważ następujący scenariusz: program poprosił o blok długości n (zamiast $n + 1$), po czym wpisał tam n znaków i zakończył ciąg zerem. Co się stanie z naszym algorytmem? Czy da się wykryć taki błąd?

Komentarz: Celem tego zdania jest przygotowanie Was do implementacji poważniejszego algorytmu zarządzania pamięcią, który będzie treścią drugiego projektu programistycznego. Potraktujcie je jako wprawkę!

²https://en.wikipedia.org/wiki/Find_first_set