Systemy operacyjne

Lista zadań nr 8

Na zajęcia 28 listopada 2024

Należy przygotować się do zajęć czytając następujące materiały: [6, rozdziały 1 – 3].

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

Zadanie 1. Na użytek przydziału i zwalniania stron pamięci w przestrzeni użytkownika systemy uniksowe udostępniają wywołania systemowe `sbrk(2)` oraz parę `mmap(2)` i `munmap(2)`. Jakie są wady stosowania `sbrk` do zarządzania rozmiarem sterty przez biblioteczny algorytm zarządzania pamięcią `malloc(3)`? Jak można to poprawić przy pomocy `mmap` i `munmap`? Kiedy procedura `free` może zwrócić pamięć do jądra?

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

Zadanie 2. Wyjaśnij różnicę między **fragmentacją wewnętrzną** i **zewnętrzną**. Czemu algorytm `malloc` nie można stosować **kompaktowania**? Na podstawie [6, §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 [6, §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 [6, §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. Funkcja «`alloc`» zwraca bloki o identyfikatorach będącymi kolejnymi literami alfabetu. Zwracane adresy są podzielne przez rozmiar słowa maszynowego. Implementacja używa dwukierunkowej listy wolnych bloków oraz boundary tags bez optymalizacji. Wyszukiwanie wolnych bloków działa zgodnie z polityką best-fit. Operacja zwalniania **gorliwie złącza** bloki i wstawia wolne bloki na koniec 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(4) alloc(8) alloc(4) alloc(4) alloc(10) alloc(6)
  free(C) free(B) free(F) alloc(6) free(D) alloc(18)
```

Czy coś by się zmieniło, gdyby algorytm wykorzystywał politykę first-fit?

Wskazówka: Wolny blok można podzielić na dwa mniejsze pod warunkiem, że obydwa mogą pomieścić węzeł listy wolnych bloków. W przeciwnym wypadku nie można tego zrobić i trzeba wziąć blok, który jest dłuższy o mały nieużytek.

Zadanie 5. Rozważmy **algorytm kubełkowy** [6, §3.6] (ang. *segregated-fit*) przydziału pamięci z gorliwym złączaniem wolnych bloków. Porównaj go z algorytmem, który zarządza jedną listą wolnych bloków zgodnie ze strategią best-fit. Jak przebiegają operacje «`malloc`» i «`free`»? Co robi «`malloc`», gdy na danej liście nie ma wolnego bloku żądanego rozmiaru? Gdzie należałoby przechowywać **węzeł strażnik** (ang. *sentinel node*) każdej z list wolnych bloków? Rozważ zastosowanie **leniwego złączania** wolnych bloków w algorytmie kubełkowym przydziału pamięci – jakie problemy zauważasz?

Ściągnij ze strony przedmiotu archiwum «so21_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 (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*), złączać (ang. *coalescing*) lub zmieniać rozmiar dwóch występujących po sobie wolnych bloków, jeśli nie da się ich złączyć. 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!

Zadanie 7. 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 odpluskwianie 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 8 (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 ffs[1] 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.

[1] https://en.wikipedia.org/wiki/Find_first_set

Literatura

[1] „*Computer Systems: A Programmer's Perspective*"
Randal E. Bryant, David R. O'Hallaron
Pearson Education Limited; 3rd edition; 2016

[2] „*Advanced Programming in the UNIX Environment*"
W. Richard Stevens, Stephen A. Rago
Addison-Wesley Professional; 3rd edition; 2013

[3] „*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*"
Michael Kerrisk
No Starch Press; 1st edition; 2010

[4] „*Systemy operacyjne*"
Andrew S. Tanenbaum, Herbert Bos
Helion; wydanie czwarte; 2015

[5] „*Operating Systems: Three Easy Pieces*"
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
https://pages.cs.wisc.edu/~remzi/OSTEP/

[6] „*Dynamic Storage Allocation: A Survey and Critical Review*"
Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles
https://www.cs.hmc.edu/~oneill/gc-library/Wilson-Alloc-Survey-1995.pdf