Systemy operacyjne

Lista zadań nr 8

Na zajęcia 3 i 11 grudnia 2019

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.

Ściągnij ze strony przedmiotu archiwum «`so19_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 4 (P). 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 5 (P, 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`[2] 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: Prawdziwy algorytm musiałby jeszcze wziąć pod uwagę strukturę pamięci podręcznej procesora.

[1] `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.275`

[2] `https://en.wikipedia.org/wiki/Find_first_set`

Zadanie 6 (P). (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!

Ściągnij ze strony przedmiotu plik «so19_projekt-shell-1.tar.gz», rozpakuj go i zapoznaj się z dostarczonymi plikami.

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

Zadanie 7 (P). Wykonywanie zewnętrznego polecenia przez powłokę uniksową ma dwa warianty. Gdy w ścieżce do pliku występuje znak ukośnika «/» to zakładamy, że użytkownik podał ścieżkę względną i uruchamiamy execve(2) bezpośrednio. W przeciwnym przypadku musimy odczytać zawartość zmiennej środowiskowej «PATH» przechowującej katalogi oddzielone znakiem dwukropka «:». Każdą ścieżkę katalogu sklejamy z nazwą polecenia (przyda Ci się procedura «strapp» z pliku «helpers.c») i wołamy «execve». Jeśli polecenie nie zostanie znalezione na dysku to «execve» wraca z błędem.

Uzupełnij ciało procedury «external_command» z pliku «command.c» zgodnie z powyższym opisem.

Wskazówka: Rozwiązanie wzorcowe liczy 10 linii. Do przetwarzania ciągów znakowych użyto procedur strndup(3) i strcspn(3).

Zadanie 8 (P). Powłoka uniksowa dzieli zadania na pierwszoplanowe (ang. *foreground job*) i drugoplanowe (ang. *background job*). Jednocześnie może istnieć tylko jedno zadanie pierwszoplanowe, natomiast zadań drugoplanowych (polecenie zakończone znakiem «&») może być wiele. Zapoznaj się z kodem odpowiedzialnym za obsługę zadań – znajduje się on w pliku «jobs.c». Przeczytaj i wyjaśnij co robią procedury «addjob», «watchjobs», «jobdone» i «killjob». Zadanie pierwszoplanowe ma zawsze numer 0.

Uzupełnij procedurę obsługi sygnału «SIGCHLD». Dla każdego zakończonego dziecka należy znaleźć odpowiedni wpis w tablicy «jobs» i wpisać mu kod wyjścia. Wykorzystaj nieblokujący wariant waitpid(2).

Wskazówka: Brakuje około 5 linii kodu.

Zadanie 9 (P). Procedura «command» w pliku «shell.c» przyjmuje tablicę ciągów znakowych reprezentującą polecenie do uruchomienia oraz rodzaj tworzonego zadania (pierwszoplanowe lub drugoplanowe). Najpierw sprawdza czy podane polecenie należy do zestawu poleceń wbudowanych (ang. *builtin command*). Jeśli nie, to przechodzi do utworzenia zadania i wykonania polecenia zewnętrznego. Po utworzeniu podprocesu i nowej grupy procesów, zadanie jest rejestrowane z użyciem «addjob». Jeśli utworzono zadanie pierwszoplanowe, to należy poczekać na jego zakończenie.

Uzupełnij ciało procedury «command» – wykorzystaj: sigprocmask(2), sigsuspend(2) i setpgid(2).

Wskazówka: Brakuje około 10 linii kodu.