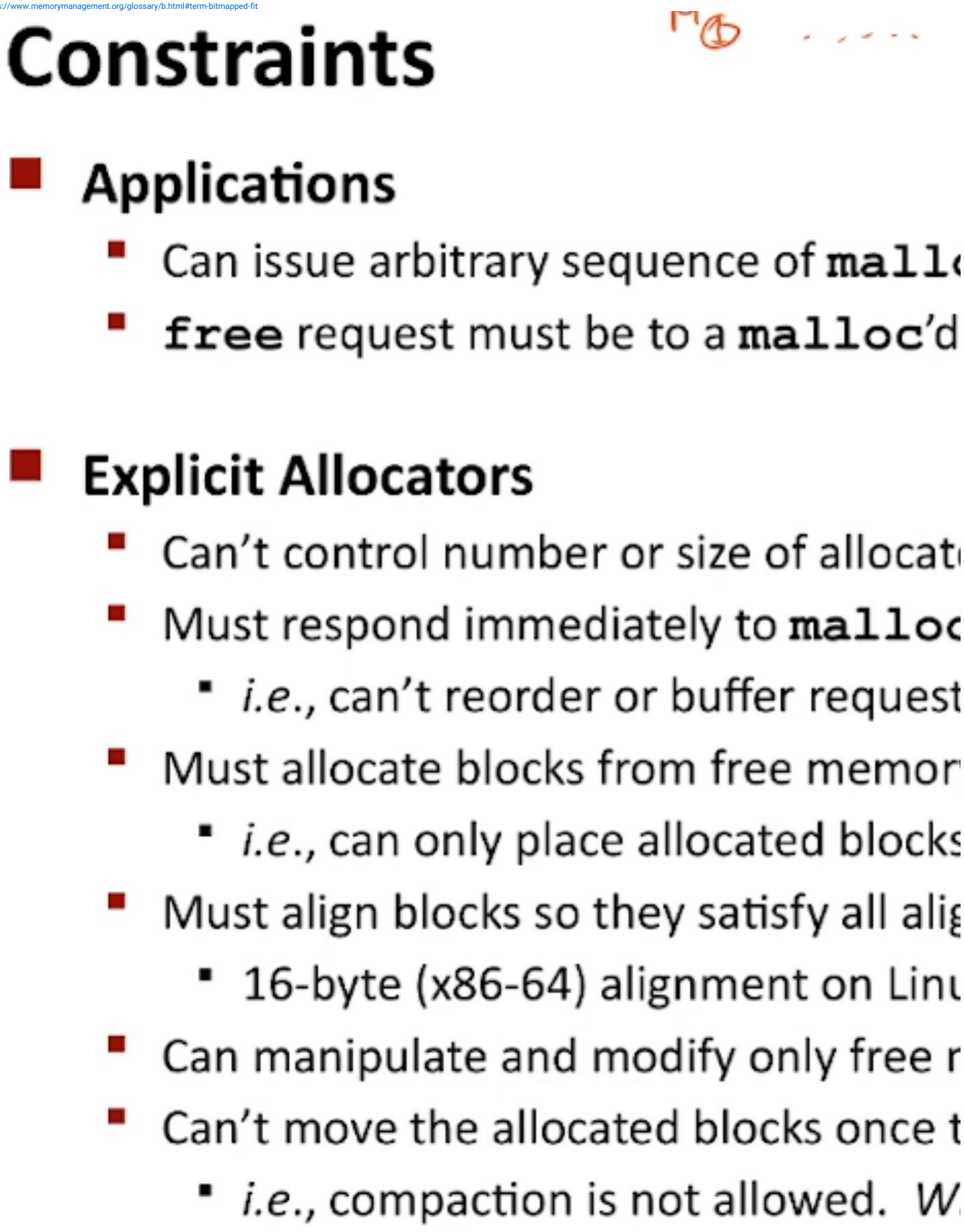
Lista 8

tags: SO

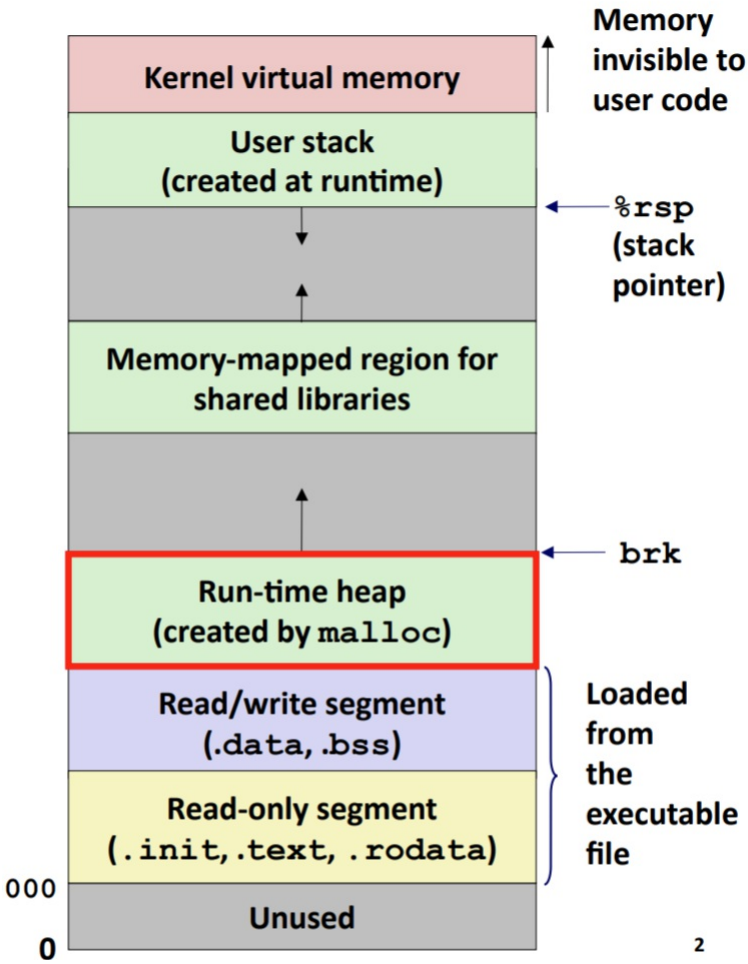
https://www.memorymanagement.org/glossary/b.html#term-bitmapped-fit



Zadanie 1

Zadanie 1. Na użytek przydziału i zwalniania stron pamięci w przestrzeni użytkow udostępniają wywołania systemowe `sbrk(2)` oraz parę `mmap(2)` i `munmap(2)`. Jakie są do zarządzania rozmiarem sterty przez biblioteczny algorytm zarządzania pamięcią `m` to poprawić przy pomocy `mmap` i `munmap`? Kiedy procedura `free` może zwrócić pam

Wskazówka: Rozważ scenariusz, w którym proces zwolnił dużo pamięci przydzielonej na początku j



::: warning

- **sbrk** – rozszerza / zwęża stertę (efektywnie alokując i zwalniając pamięć procesu)
- **mmap(*addr, length, ...)** – odwzorowuje plik w pamięć
- **munmap** – zwalnia pamięć zaalokowaną przez mmap :::

> For very large requests, malloc() uses the mmap() system call to find addressable memory space. *This process helps reduce the negative effects of memory fragmentation* when large blocks of memory are freed but locked by smaller, more recently allocated blocks lying between them and the end of the allocated space. In this case, in fact, had the block been allocated with brk(), it would have remained unusable by the system even if the process freed it. (emphasis mine)

Używając sbrk Jak widać, nie możemy przesunąć brk (czyli pamięć jest zwalniana tylko symbolicznie), bo mały fragment z prawej musi pozostać zaalokowany, czyli gdybyśmy chcieli teraz wczytać jakiś większy blok pamięci to musielibyśmy rozszerzać stertę o co najmniej jego rozmiar, marnując wcześniej "zwolnioną" pamięć. Widać tutaj również, że free zwraca pamięć do jądra tylko w wypadku gdy zmniejszając `brk` miniemy w całości zaalokowany blok pamięci.

Używając mmap Tutaj zwolniony blok pamięci rzeczywiście znika z pamięci procesu.

Zdanie 2

Zadanie 2. Wyjaśnij różnicę między **fragmentacją wewnętrzną** i **zewnętrzną**. C nie można stosować **kompaktowania**? Na podstawie [6, §2.3] opowiedz o dwóch występowania fragmentacji zewnętrznej.

::: info

- **fragmentacja wewnętrzna** – pamięć efektywnie niewykorzystana przez użytkownika powstająca na skutek:
  - wyrównywania do wielokrotności 2 słów
  - wydzielania cześci pamięci dla struktury danych nią zarządzającą
  - przestrzegania pewnych polityk
- **fragmentacja zewnętrzna** – ma miejsce, gdy roszerzamy stertę mimo, że suma rozmiarów wolnych bloków jest wystarczająca do pomieszczenia nowych danych (ale żaden blok z osobna już nie). :::

Na podstawie §2.3 opowiedz o dwóch głównych przyczynach występowania fragmentacji zewnętrznej.

- **Fragmentation is caused by isolated deaths.** A crucial issue is the creation of free areas whose neighboring areas are not free. This is a function of two things: which objects are placed in adjacent areas and when those objects die. Notice that if the allocator places objects together in memory, and they die at the same time" (with no intervening allocations), no fragmentation results: the objects are live at the same time, using contiguous memory, and when they die they free contiguous memory. An allocator that can predict which objects will die at approximately the same time can exploit that information to reduce fragmentation, by placing those objects in contiguous memory

::: warning Czyli chcemy stawiać bloki podobnego przeznaczenia obok siebie w nadziei, że zostaną zwolnione po kolei tworząc większy fragment ciągłej pamięci. :::

- **Fragmentation is caused by time-varying behavior.** Fragmentation arises from changes in the way a program uses memory - for example, freeing small blocks and requesting large ones. This much is obvious, but it is important to consider patterns in the changing behavior of a program, such as the freeing of large numbers of objects and the allocation of large numbers of objects of different types. Many programs allocate and free different kinds of objects in different stereotyped ways. Some kinds of objects accumulate over time, but other kinds may be used in bursty patterns. The allocator's job is to exploit these patterns, if possible, or at least not let the patterns undermine its strategy.

Zdanie 3

Zadanie 3. Posługując się wykresem wykorzystania pamięci w trakcie życia proc wzorcach przydziału pamięci występujących w programach [6, §2.4]. Na podstawie pa „*Exploiting ordering and size dependencies*" wyjaśnij jaki jest związek między czas rozmiarem? Wyjaśnij różnice między politykami znajdowania wolnych bloków: **first** Na podstawie [6, §3.4] wymień ich słabe i mocne strony.

Rozróżniamy trzy najczęstsze wzorce przydziału pamięci:

- **Ramp** – Many programs accumulate certain data structures monotonically over time This may be because they keep a log of events or because the problemsolving strategy requires building a large representation after which a solution can be found quickly

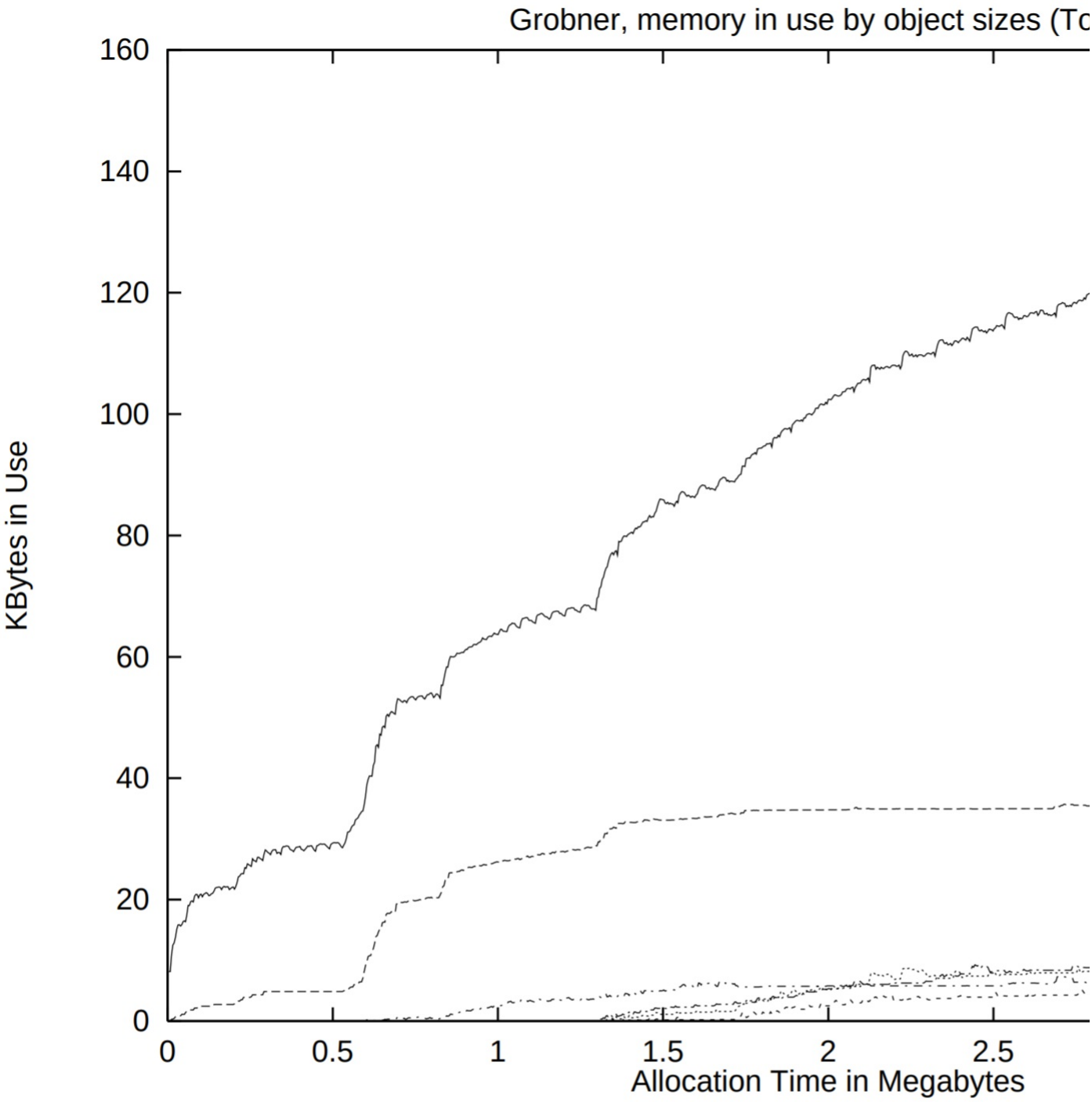
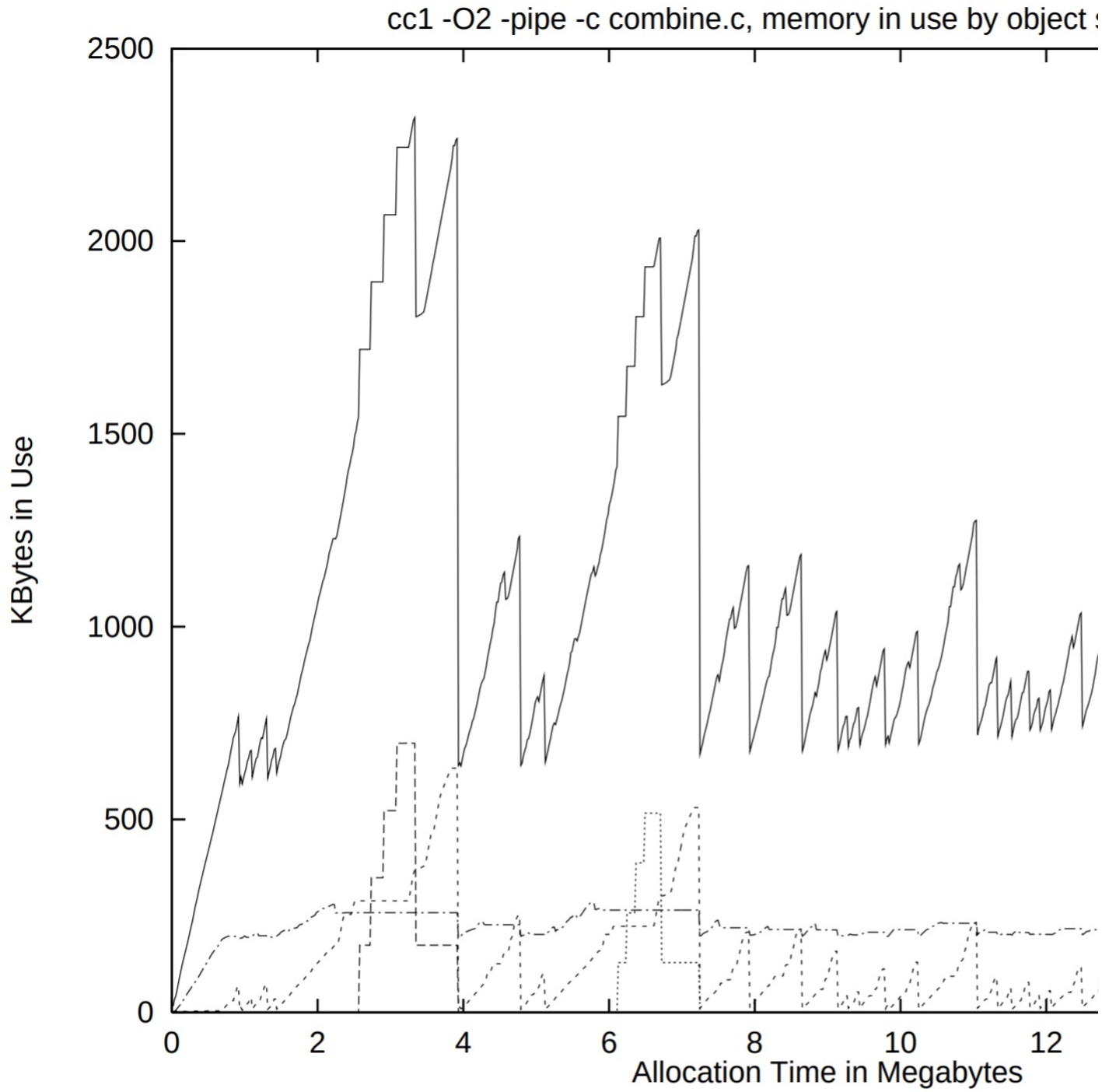


Fig. 2. Profile of memory usage in the Grobner pro

- **Peaks** – Many programs use memory in bursty patterns building up relatively large data structures which are used for the duration of a particular phase and then discarding most or all of those data structures. Note that the surviving data structures are likely to be of different types because they represent the results of a phase as opposed to intermediate values which may be represented dierently. A peak is like a ramp but of shorter duration.



- **Plateau** – Many programs build up data structures quickly and then use those data structures for long periods often nearly the whole running time of the program.
-

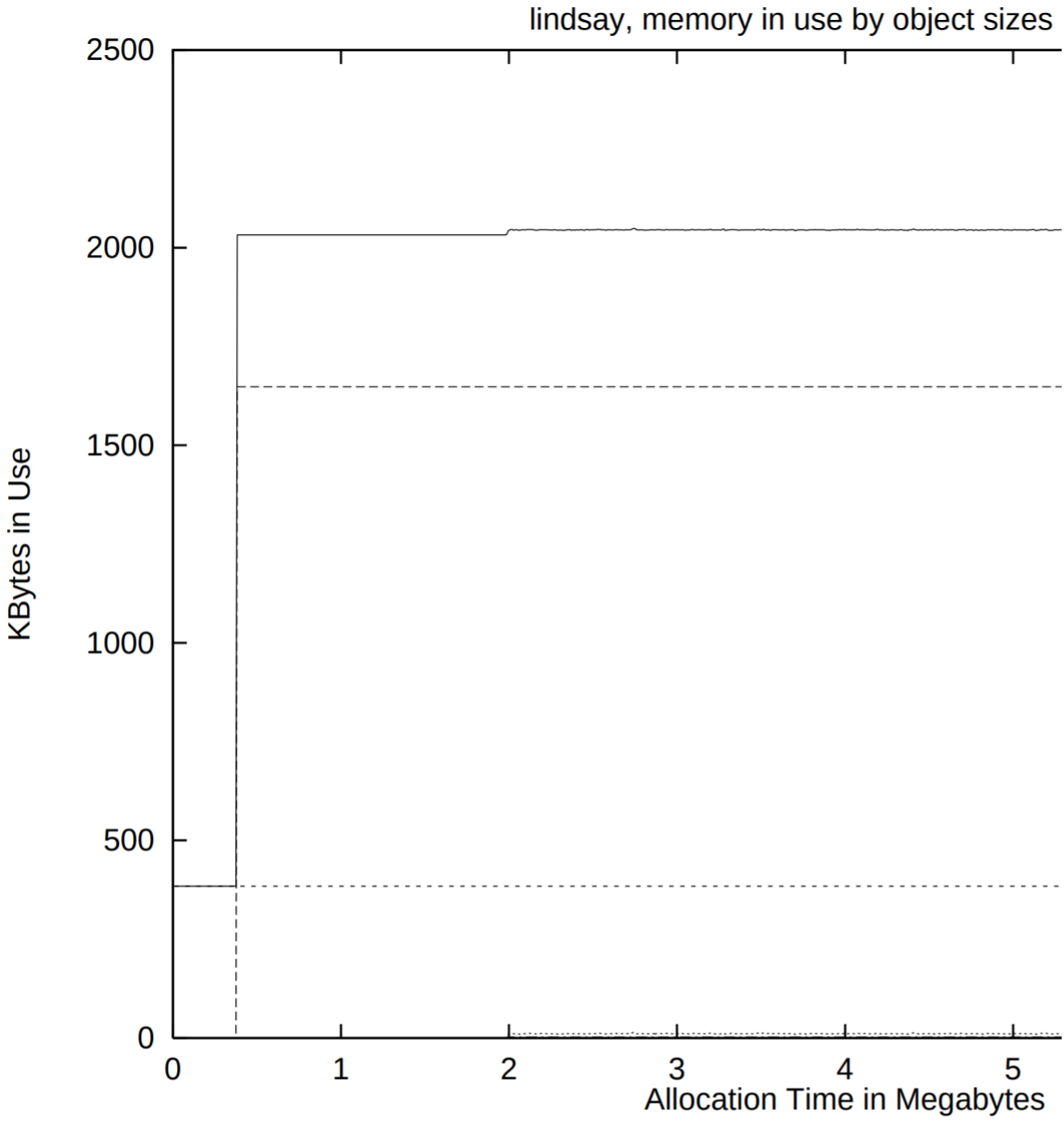


Fig. 3. Profile of memory usage in Lindsay's hyperc

Jaki jest związek czasu życia bloku z jego rozmiarem?

Po obiektach różnych rozmiarów spodziwamy się różnych typów, a zatem również różnego przeznaczenia. Zwykle większy blok pamięci wiąże się więc z dłuższym czasem życia podczas, gdy małe bloczki żyją krótko.

Polityki znajdowania różnych bloków

- **first-fit** – przeglądamy kolejne bloki, alokujemy w pierwszym możliwym miejscu (rozwiązanie jest proste, ale prowadzi do odkładania się małych bloczków na początku listy bloków, które prowadzą do jej puchnięcia, a stąd wolniejszych wyszukiwań)
- **next-fit** – wykonujemy first-fit, ale zaczynamy w miejscu, w którym ostatnio skończyliśmy (rozwiązuje problem poprzedniej polityki, ale prowadzi do przeplatania się bloków z różnych faz programu, może również popsuć jego lokalność)
- **best-fit** – znajdujemy najmniejszy wolny blok mieszczący nasze dane (w ogólnym przypadku dobre rezultaty, problemy z optymalizacją złożoności czasowej i skutecznością dla dużych stert)

Zdanie 4

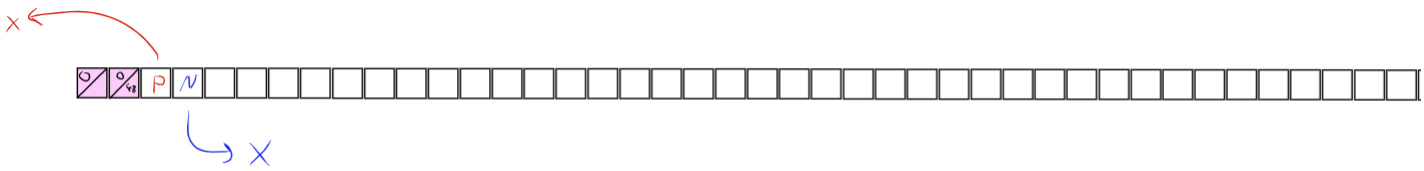
Zadanie 4. Algorytm przydziału pamięci udostępnia funkcje o sygnaturach «`alloc: id -> void`» i ma do dyspozycji obszar 50 słów maszynowych. Funkcja «`alloc`» zwr rach będącymi kolejnymi literami alfabetu. Zwracane adresy są podzielne przez rozm Implementacja używa dwukierunkowej listy wolnych bloków oraz boundary tags bez wanie wolnych bloków działa zgodnie z polityką best-fit. Operacja zwalniania **gorliwie** wolne bloki na koniec listy. Posługując się diagramem z wykładu wykonaj krokow przydziału pamięci dla poniższego ciągu żądań. Należy wziąć pod uwagę miejsce zaj danych algorytmu przydziału oraz nieużytki.

```
alloc(4) alloc(8) alloc(4) alloc(4) alloc(10) alloc
  free(C) free(B) free(F) alloc(6) free(D) alloc(18
```

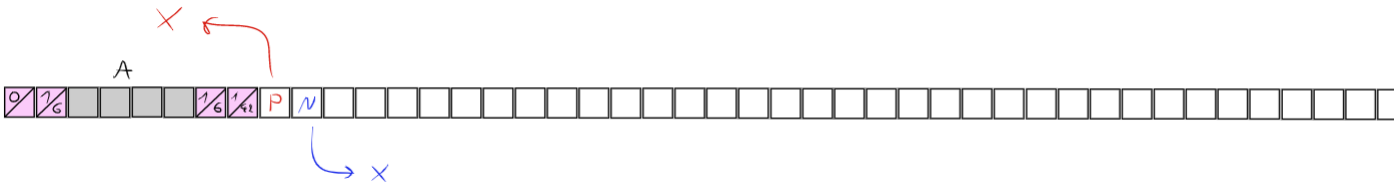
Czy coś by się zmieniło, gdyby algorytm wykorzystywał politykę first-fit?

Wskazówka: Wolny blok można podzielić na dwa mniejsze pod warunkiem, że obydwa mogą po bloków. W przeciwnym wypadku nie można tego zrobić i trzeba wziąć blok, który jest dłuższy o ma

https://idroo.com/board-5C07vJLEdK

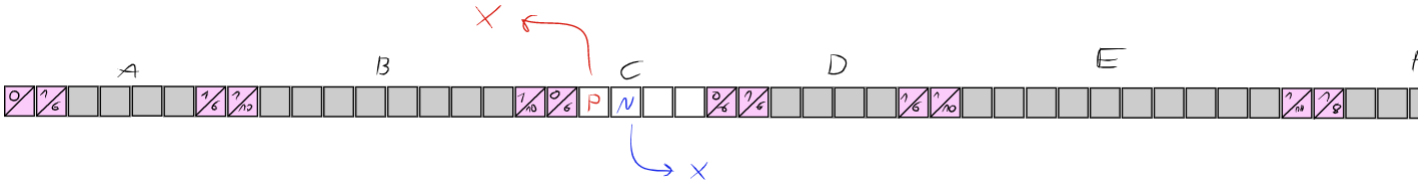


alloc(4) -> A

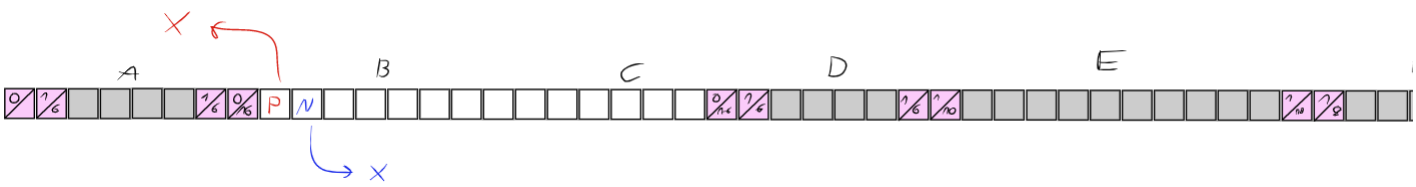


alloc(8, 4, 4, 10, 6) -> B, C, D, E, F

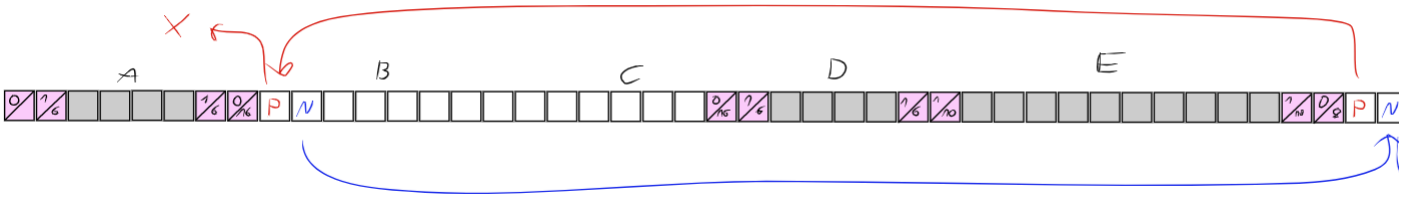
free(C)



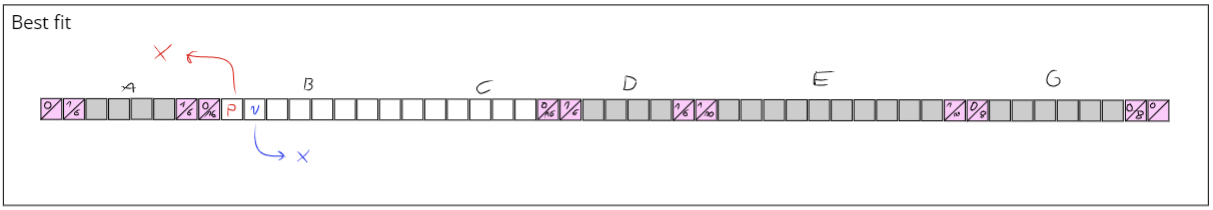
free(B)

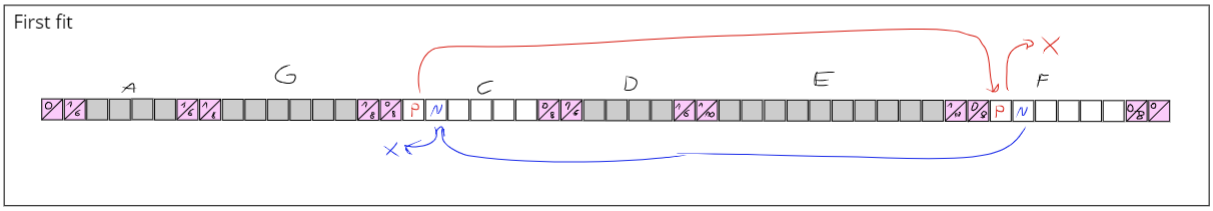


free(F)

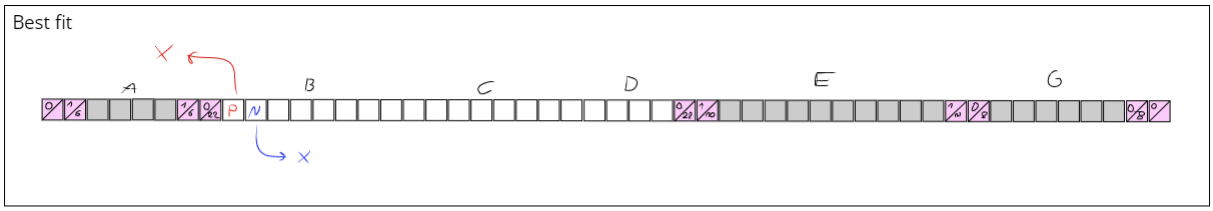


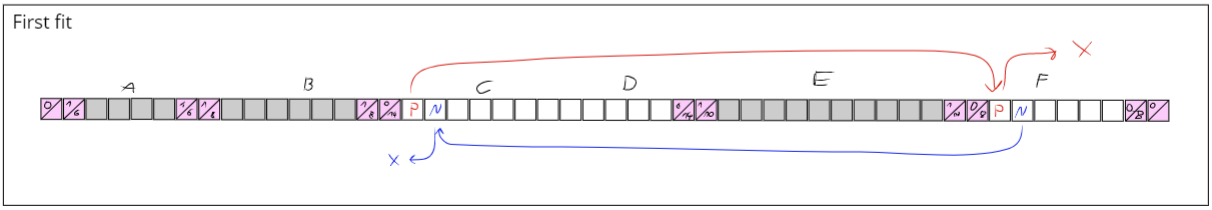
alloc(6) -> G



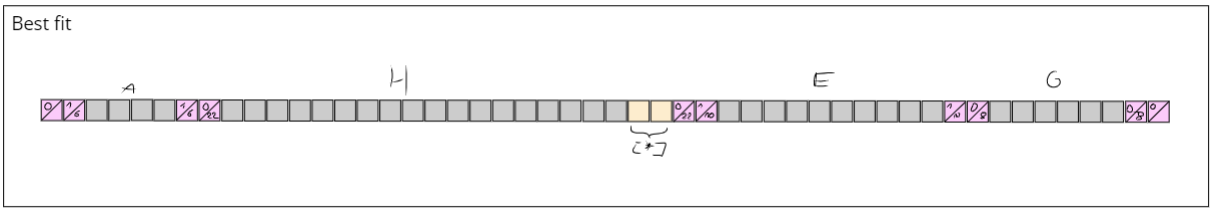


free(D)



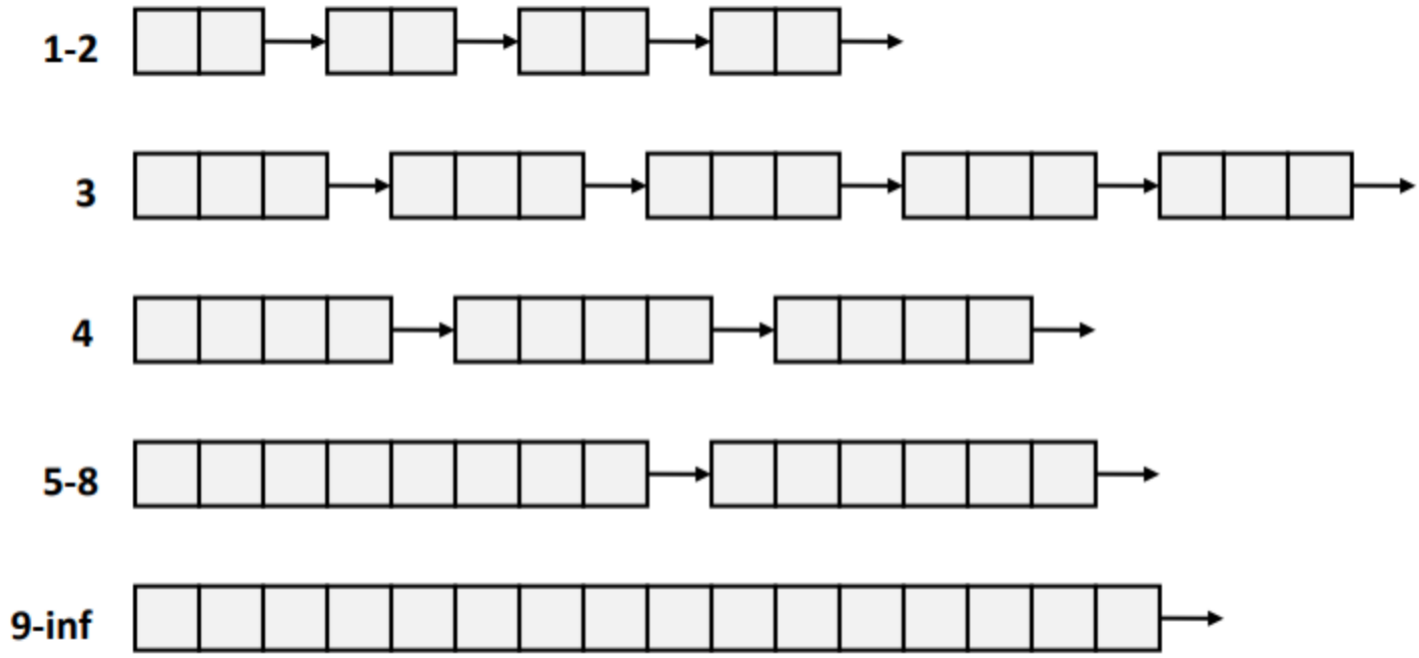


alloc(18) -> H



Zdanie 5

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



:::info ***algorytm kubełkowy***

- malloc -- zaglądamy do kubełka o odpowiednim rozmiarze, jeśli jest niepusty do alokacji używamy najmniejszego znajdującego się w nim bloku. W przeciwnym przypadku algorytm próbuje znaleźć najmniejszy blok w kolejnych kubełkach który następnie zostaje podzielony, a jego niewykorzystana część jest umieszczana z powrotem w tablicy kubełków. Jeśli odpowiedniego bloku nie udało się znaleźć pozyskiwana jest dodatkowa pamięć od systemu operacyjnego.
- free -- jeśli możliwe magicznie scalamy zwalniany blok z jego sąsiadami i umieszczamy go w odpowiednim kubełku (tutaj wypdałoby usunąć sąsiadów z tablicy kubełków jakoś...)

Algorytm jest bardzo podobny do best-fit na pojedynczej liście, jako że wybieramy najmniejszy możliwy blok do alokacji. Dużą zaletą takiego algorytmu jest logarytmiczny czas działania. :::

Gdzie należałoby przechować strażnika?

::: danger ***(kupon na darmowe pytanie w akcji)*** Takie rozwiązanie zdaje się nie wymagać cykliczności wolnych list, więc przypuszczalnie wystarczyłby jeden strażnik dla całej struktury, gdzie ostatni element każdej listy wskazywałby na tego strażnika? :::

Jakie są problemy wynikające z leniwego złączania?

:::info **leniwe złączanie** – łączymy bloki tylko, gdy brakuje nam bloku dość dużego by dokonać alokacji (w przeciwieństwie do łączenia przy każdym zwalnianiu) :::

Motywacją takiego rozwiązania byłoby przeciwdziałanie zjawisku wydłużających się czasów wyszukiwań dla małych alokacji (jeśli bloki są łączone leniwie to często kubełki dla małych bloków są puste i ciężar tablicy gromadzi się w przedziałach dla większych bloków). Niestety, takie rozwiązanie rezultuje zwiększoną fragmentacją. Intuicyjnie, duże bloki są lepsze bo możemy dzielić je w miarę potrzeb. Działając na większej liczbie mniejszych bloków mamy mniejszą kontrolę nad przebiegiem fragmentacji.

Zdanie 6

Ściągnij ze strony przedmiotu archiwum «so21_lista_8.tar.gz», następnie rozpakuj i zapoznaj si

UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczone w koment

UWAGA! Dla metod przydziału pamięci użytych w poniższych zadaniach należy być przygotowany

- 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 probleme

Zadanie 6 (2). (Implementację zadania dostarczył Piotr Polesiuk.)

Program «stralloc» implementuje algorytm zarządzania pamięcią wyspecjalizowan miejsca dla ciągów znakowych nie dłuższych niż «MAX_LENGTH». Ponieważ algorytm składowane ciągi znakowe, to nie musi dbać o wyrównanie adresu zwracanego przez

Podobnie jak w programie «objpool» będziemy zarządzać pamięcią dostępną za nagłó tym zakodujemy **niejawną listę** (ang. *implicit list*) jednokierunkową, której węzły są bajcie bloku. Wartość bezwzględna nagłówka bloku wyznacza jego długość, a znak c to czy blok jest wolny, czy zajęty. Nagłówek bloku o wartości zero koduje koniec lis arena ma długość 65536 bajtów to procedura «init_chunk» musi wypełnić zarz blokami nie większymi niż «MAX_LENGTH+1».

Twoim zadaniem jest uzupełnienie brakujących fragmentów procedur «alloc_block» z nich jest zdecydowanie trudniejsza i wymaga obsłużenia aż pięciu przypadków. Bę (ang. *splitting*), złączać (ang. *coalescing*) lub zmieniać rozmiar dwóch występują bloków, jeśli nie da się ich złączyć. Druga procedura jest dużo prostsza i zaledw upewniwszy się wcześniej, że użytkownik podał prawidłowy wskaźnik na blok.

Przed przystąpieniem do rozwiązywania przemyśl dokładnie działanie procedur. Pomy lezienia. Jedyną linią obrony będzie tutaj obfite sprawdzanie warunków wstępnych fu

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

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

Zadanie 7. Program «objpool» zawiera implementację **bitmapowego przydział** miarze. Algorytm zarządza pamięcią w obrębie aren przechowywanych na jednokieru Pamięć dla aren jest pobierana od systemu z użyciem wywołania mmap(2). Are przechowujący węzeł listy i dodatkowe dane algorytmu przydziału. Za nagłówkiem ar na metadane, a także bloki pamięci przydzielane i zwalniane funkcjami «alloc_blo

Używając funkcji opisanych w bitstring(3) uzupełnij brakujące fragmenty proced bitmapy są przechowywane za końcem nagłówka areny. Ponieważ odpluskwianie algo należy korzystać z funkcji assert(3) do wymuszania warunków wstępnych procedu algorytmu zarządzania pamięcią musi przechodzić test wbudowany w skompilowany

Zdanie 8

Zadanie 8 (bonus). Zoptymalizuj procedurę «alloc_block» z poprzedniego zada niskiej wydajności jest użycie funkcji «bit_ffc». Należy wykorzystać dwa sposob (a) użycie jednocyklowej instrukcji procesora ffs[1] wyznaczającej numer pierwszego us maszynowym (b) użycie wielopoziomowej struktury bitmapy, tj. wyzerowany i-ty bit mówi, że w i-tym słowie maszynowym bitmapy poziomu $k+1$ występuje co najmnie

Komentarz: Bardzo dobry algorytm musiałby jeszcze wziąć pod uwagę strukturę pamięci podręczn