Technisch Rapport

Geheugenallocatie in C

6 mei 2015

Students:
Kjeld Oostra
10748598

Steven Raaijmakers
10804242

Docent:
R. Poss

Cursus:
Besturingssystemen

Inhoudsopgave

1 Inleiding

Één van de functies van een besturingssysteem is het alloceren van geheugen voor programma's. In C bestaat hiervoor een functie, die in de modules - bestaande uit banken - binnen het geheugen de nodige geheugenbanken activeert of deactiveert zodra deze moeten worden benut. Het doel van deze opdracht was een dergelijk systeem te ontwikkelen voor binnen een simulatie-omgeving, welke de benoemde taak zo efficiënt en duurzaak mogelijk moet uitvoeren. In dit rapport wordt uitgelegd welke methode wij hebben geïmplementeerd, welke resultaten dit heeft opgeleverd en welke conclusie wij hieruit hebben getrokken.

2 Doelen

- if you have submitted an archive in the right format with an AUTHORS file and a non-empty report.pdf
- if your source code builds without errors and you have modified mm.c in any way
- if your code supports a simple sequence of calls to mm_alloc without errors using only 1 memory module and returning a valid range every time
- if your overhead per allocated byte is asymptotically constant in long sequences of calls to mm_alloc followed by mm_free using only 1 memory module (in other words, your mm_free causes banks to be deactivated eventually
- if your code can optimize for performance: during sequences of mm_alloc then mm_free, hw_activate is called less often than there are calls to mm_alloc (bank reuse)
- if your code has a significantly lower number of ällocations failures despite empty space"than the dumb allocator
- if your report satisfies the requirements set below
- if your manager can satisfy multiple calls to mm_alloc by sharing a common bank (when the allocation size is smaller than a bank).
- if your code can optimize for energy usage by deactivating banks released by mm_free
- no complaints by gcc and vangrind

- if your code tries to increase locality, i.e. mm_alloc tries to reuse the banks most recently deallocated by mm_free
- maximum performance

3 Implementatie

Een module bestaat uit banken, alle met dezelfde grootte. Wanneer een stuk geheugen gealloceerd moet worden, moet hiervoor ruimte gevonden worden binnen één module. Binnen een module wordt dan een zogenaamde stuk geheugen gereserveerd, een *chunk*.
Onze implementatie staat toe dat een chunk over meerdere banken verspreid staat, waardoor het tevens mogelijk is om juist meerdere chunks per bank te hebben. Elke chunk heeft een klein stukje administratie (genaamd een *head*), waarin staat waar de vorige en volgende chunk zich bevinden, en hoe groot de huidige chunk is.
Ook bevat elke module een aparte header, die onder andere informatie bevat over welke banken er aanstaan in de gehele module én verwijst naar de volgende module.
Bij initialisatie van een module wordt er een module-header geplaatst, met meteen hierna een chunk-header. De chunk-header bevat op dat moment de informatie over de daaropvolgende ruimte in de module. Bij de initialisatie zal dat dus de volledige ruimte zijn, minus de grootte van de module-header en de chunk-header.
Wanneer er een stuk geheugen moet worden gereserveerd, wordt er een nieuwe chunk-header geplaatst aan het einde van dat stuk geheugen. Een chunk wordt dus als het ware opgesplitst, waarbij de nieuwe chunk-header geplaatst wordt in de huidige chunk. De grootte van de nieuwe chunk zal dus de grootte van de oude chunk minus de grootte van het stuk geheugen dat moet worden gereserveerd.
Aangezien alle relevante informatie in de administratie van de chunks en van de modules wordt opgeslagen, zijn er weinig zoek-algoritmes nodig om de juiste gegevens te verkrijgen. Dit heeft een kleine impact op de ruimte in het geheugen, maar heeft als grote voordeel dat de asymptotische complexiteit overal maximaal $\mathcal{O}(n)$ is.
Bij het legen van de chunks wordt direct gekeken of de bank(en) waarin de chunk zich bevindt na de leging helemaal leeg is. Indien dit het geval is, wordt deze gedeactiveerd nadat de bank binnen 10 operaties niet opnieuw geactiveerd hoeft te worden.
Alle chunks maken onderdeel uit van een *double LinkedList*. Elke chunk-header verwijst namelijk naar een vorig- en volgend chunk-element. Bij het verwijderen vn een chunk, wordt er feitelijk alleen maar gezorgd dat de linker-chunk(header) en de rechter-chunk(header) van het te-verwijderen

element voortaan naar elkaar verwijzen, dus zonder tussen komst van de te-verwijderen header. De modules maken onderdeel van een *single LinkedList* uit, de modules verwijzen onderling namelijk naar.

4 Doelstellingen

Ons doel was om een geheugenallocatie te schrijven die een betere perfomance leverde dan de dumb-allocator, en tevens ook minder stroom verbruikte.

5 Resultaten

simple.t

Performance (op / s)	26.290
Energy efficieny (op / J)	4.154

6 Conclusie

Een vergelijking met de 'domme'-implementatie met input file simple.t:

	Onze implementatie	Domme implementatie
Performance (op / s)	26.290	14.778
Energy efficieny (W)	4.154	2.613

Onze implementatie kan bijna twee keer zo veel operaties per seconde verwerken. Wat betreft energiegebruik is onze implementatie echter wat minder efficiënt dan de domme implementatie, waarschijnlijk aangezien er meer banken worden gebruikt en deze niet altijd even consistent worden gedeactiveerd.