KTH ICT

Laboration 3

Minneshantering v. 4

ID2206/ID2200 (fd. 2G1520/2G1504)
Operativsystem

Uppdaterad 2012

Minneshantering

ID2206/ID2200 (fd. 2G1520/2G1504) Operativsystem

1.0 Bakgrund

Biblioteksanropen `malloc(3)`, `free(3)` och `realloc(3)` tillkom i och med UNIX Version 7, p g a att man önskade ge programmeraren ett verktyg för att förvalta systemets minnesresurser oberoende av operativsystem och datorarkitektur. Den här laborationen skall visa hur man använder ett antal biblioteksrutiner för att överbrygga gapet mellan operativsystemets maskinnära och programmerarens abstrakta sätt att hantera minne. I den här laborationen kommer du att själv få implementera de ovannämnda funktionerna. Uppgiften är medvetet löst formulerad och har karaktären av ett litet självständigt projekt som en nyutexaminerad civilingenjör skulle kunna få som ”uppvärmningsprojekt” i industrin. Förutom att du själv har full frihet att välja algoritmer och datastrukturer i din implementation, skall du också genomföra en *grundlig* utvärdering av de algoritmer du implementerat jämfört med systemets funktioner för minneshantering. Inom ramen för uppgiften finns många möjligheter för den intresserade att göra optimeringar och utvidgningar. Dock kan man med även de enklare algoritmerna komma i närheten av prestandan hos biblioteksimplementationerna av `malloc()`.

2.0 Uppgiften

2.1 Ramformulering

Skriv ett litet bibliotek med funktionerna `malloc(3)`, `free(3)` och `realloc(3)`, som kan användas istället för de fördefinerade funktionerna i standardbiblioteket libc[1]. Till din hjälp har du C-biblioteksfunktionen `mmap(2)`. Tidigare använde man ett systemanrop `sbrk(2)` (`sbrk` är ett skal kring systemanropet `brk(2)`) med vars hjälp man kunde öka/minska storleken på heapen. Den gav ett intuitivt enklare gränssnitt och du kan fortfarande se hur koden såg ut när man använde `sbrk(2)`. Att man valt att gå över till `mmap(2)` beror på att man i flertrådade program har behov av att inte bara kunna allokera mer minne på heapen utan också att man vill kunna skapa och eventuellt utöka en stack per tråd. med `mmap(2)` kan man ange var i det virtuella minnet en area skall reserveras och också ange rättigheter till arean (skriv, läs och exekvering). Att man kan ange var en minnesarea skall reserveras gör också att man klarar att hantera virtuella adressrymder som inte är kontinuerliga, där t.ex delar av operativsystemet mappas in i processens virtuella minnesrymd.

2.2 Deluppgifter, obligatoriska

1. Dina funktioner skall ge mera minne så länge man kan reservera mera minne via `mmap(2)`.
2. Gränssnittet skall följa ANSI/ISO-standard. (Sektion 3.0).
3. Dina funktioner skall inte förbruka minne, d v s av `free(3)` tillbakalämnade minnesdelar skall kunna delas ut igen vid kommande `malloc(3)`.

1. Biblioteket heter /usr/lib/libc.* och länkas vanligtvis med vid alla kompileringar av C-program. Normalt extraheras och länkas in endast den/de objektmoduler från libc.* som svarar mot de biblioteksfunktioner som C-programmet använder.

4. Du skall i din rapport förklara all den kod i `malloc.c` som endast kompileras om macrot `MMAP` är definerat. Och du ska dessutom svara på följande frågor:
   - I koden intialiseras `__endHeap` med hjälp av `sbrk(2)` vilket inte är så snyggt och inte skulle fungera på system där `sbrk(2)` inte stöds. Vad händer om du tar bort dessa intialiseringar (två `if`-satser)? Vad skulle `__endHeap` då representera?
   - Vad händer om man skickar `NULL` som första parameter till `mmap(2)`? Vad fungerar/fungerar inte? Vet man då var minnet reserveras?
   - Vad händer om man byter ut/tar bort flaggan `MAP_SHARED` i anropet till `mmap(2)`?
5. Implementera flera olika minnesallokerings metoder. För kursen ID2206 (2G1520) skall samtliga fyra minnesallokeringsmetoder implementeras och utvärderas. För den mindre kursen ID2200 (2G1504) skall First-fit och minst en ytterligare metod implementeras och utvärderas. De fyra metoderna är:
   - First-fit, Best- och Worst-fit
   - Quick-fit, som är en typ av optimering (se nedan). För quick fit gäller att antalet "snabb-listor" skall vara parametriserbart och styras via ett pre-processor makro (symbol) NRQUICKLISTS Om NRQUICKLISTS exempelvis definierats att vara 3 skall snabblistor med blockstorlekarna 8, 16 och 32 bytes stora block skapas.
   Man skall kunna välja metod med hjälp av att ge symbolen `STRATEGY` ett lämpligt värde vid kompileringen (se nedan). Du skall jämföra maximalt använt *systemminne* och *exekveringstid* för dessa fyra algoritmer och även jämföra med systemets `malloc(3)`. Resultaten skall du redovisa m h a diagram/tabell (Sektion 4.2).
6. Implementera hopslagning av lediga minnesblock till större vid behov i First/Best/Worst-fit. Redogör för metoden och varför du har valt just denna, lagt anropet till hopslagningsrutinen just där du har gjort mm. (Minnessnål eller snabb eller...).
7. Förbättra ANSI/ISO- standarden genom att returnera NULL då man begär en minnesarea om noll (0) bytes
8. (för ID2206) ***Förberedelsefråga:*** *Hur mycket minne slösas bort i medel och i värsta fallet i de block som allokeras via malloc() ur "Quick fit" listorna? (ge svaret i procent)*
9. Gör en noggrann utvärdering av dina algoritmer. Förklara varför du valt utvärderingsmetoden (vilken måste vara realistisk) och hur du designat de program du använder för utvärderingen.

2.3 Quick fit - en möjlig optimering

För att optimera en minneshanterare krävs att man förstår tillämpningarnas krav på minneshanteraren. Utgående från de kraven kan man sedan försöka optimera minneshanteraren med avseende på exekveringshastighet och/eller minneutnyttjande. Tyvärr är det ofta svårt, för att inte säga omöjligt, att optimera i bägge avseendena.

Det finns i princip två vanliga beteenden hos applikationer:

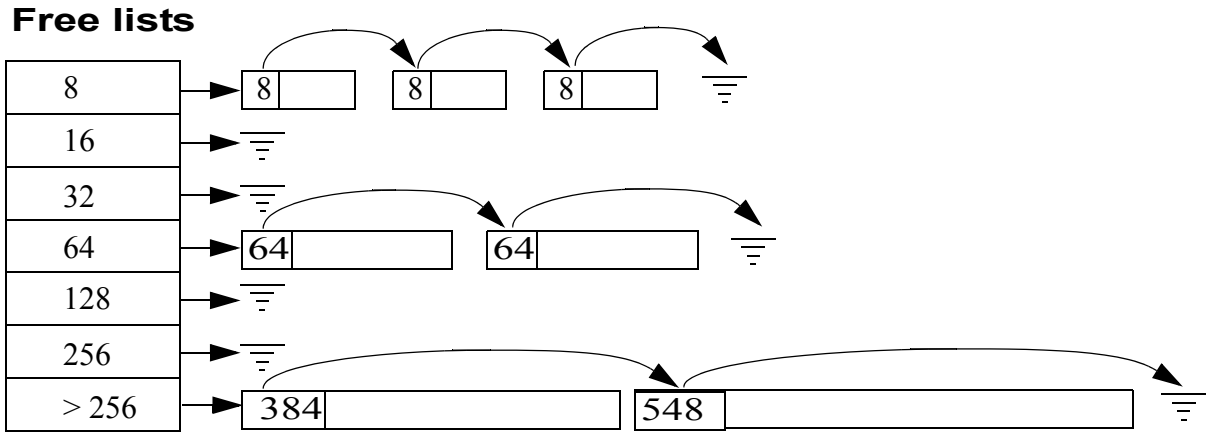
- applikationen allokerar ett (mindre) antal stora datablock
- applikationen allokerar många små datablock av ett fåtal storlekar

Det första fallet handlar ofta om tillämpningar där man allokerar minne för dataareor som används för numeriska beräkningar, bildbehandling etc. I de fallen rör det sig oftast inte om så många allokeringar/avallokeringar och enkla algoritmer som First-fit fungerar oftast bra.

I det andra fallet, vilket kanske är det vanligaste, handlar det om program som allokerar objekt för att t.ex. bygga länkade listor eller träd. Det är vanligt förekommande i olika typer av språk vare sig de är objektorienterade som C++, JAVA, imperativa som C eller i andra typer av språk som Prolog, Lisp etc. I de senare fallen märker man ofta inte som användare att objekten/minnesare-

orna skapas/allokeras under exekveringen.

För det här fallet kan man göra en förhållandevis enkel prestandaoptimering som i sin enklaste form optimerar enbart exekveringshastigheten på bekostnad av minnesutnyttjandet. Iden är att man för att hantera allokeringar av små block håller flera listor med lediga minnesblock, där varje lista innehåller block av samma storlek. Vid allokering väljer man ett block ur den lista som har minsta storlek som rymmer den storlek som begärs via t.ex. `malloc()`. I exemplet i figuren nedan skulle man t.ex. ta ett block ur 64 bytes listan om via `malloc()` begär ett block om exempelvis 44 bytes. Man delar inte upp minnesblocket innan man lämnar ut det via `malloc()`, utan slösar lite med minnet för att få det hela att gå så fort som möjligt. Därav namnet "quick fit". På samma sätt slår man heller inte ihop intilliggande block då de återlämnas, utan de länkas in direkt i respektive lista. För att klara godtyckliga större allokeringar har man en vanlig First/Best/Worst fit lista för allokeringar över en viss storlek. Figuren nedan visar en schematisk bild av Quick Fit.



FIGUR 1. Quick fit

Ofta väljer man blockstorlekarna i "Quick fit" listorna (de sex första i exemplet ovan av "free" listorna) som jämna tvåpotenser. Avvägningar man måste göra är vilken minsta blockstorlek man skall ha (i exemplet ovan 8 bytes) och vilken den största blockstorleken skall vara (i exemplet ovan 256 bytes). Sista listan i exemplet ovan är en "vanlig" first fit implementation för det fall man allokerar block större än 256 bytes.

Alla fri-listorna är tomma från början och allokeras först vid behov på heapen av minneshanteraren. I exemplet ovan initierar man t.ex. 64 bytes listan med ett antal element första gången man begär att allokera en minnesarea via t.ex. `malloc()` med en storlek mellan 33 och 64 bytes. Initieringen går normalt till så att man skapar ett lämpligt minnesutrymme på heapen via anrop till `brk()` eller `sbrk()`, t.ex. kan man allokera en area med storleken hos en (eller flera) sida. I det utrymmet bygger man sedan direkt en lista av fria block av samma storlek (i vårt exempel 64 bytes) som länkas in i fri-listan. Har du lyckats implementera First-fit är den här algoritmen ganska enkel att implementera, speciellt som du bör kunna använda samma datastrukturer för att bygga dessa listor som du gör i fallen First/Best/Worst fit.

När vi ändå håller på med optimeringar kan man fråga sig om man kan man förbättra "Quick fit" ytterligare? En möjlig förbättring är att överväga att bestämma storleken på blocken i quick fit listorna adaptivt under programmets körning. Om man till exempel upptäcker att man ofta allokerar block av storleken 82 bytes skulle man kunna skapa en speciell lista för det fallet. Innan man ger sig in i den här typen av ytterligare optimeringar bör man dock vara säker på att man verkligen vinner något på det och inte bara skapar ytterligare komplexitet. Dvs. det är inget du förväntas göra i den här labbuppgiften och heller inget som man normalt skulle göra.

3.0 Gränssnitt

Gränssnittet skall vara utformat som beskrivet i ANSI/ISO-standard under punkterna 7.10.3.2, 7.10.3.3 och 7.10.3.4 (se Sektion 7.0 sist i labbpeket). Om du vill jämföra med systemets olika `malloc(3)` (som inte följer standardspecifikationerna) så hänvisas till manbladet. (Vid det här laget är det hög tid att göra ”`man -s 3c malloc`”!).

Typdeklarationerna som används i standarden, skall även användas av dig. Du hittar deklarationerna i include-filerna: `/usr/include/stdlib.h` och `/usr/include/unistd.h`.

Dessa inkluderas med

```
#include <stdlib.h> respektive #include <unistd.h>
```

För att ditt bibliotek skall vara komplett skall du tillhandahålla följande filer med innehåll:

- `malloc.h:` Prototypdeklarationer för funktionerna `malloc(3)`, `free(3)`, och `realloc(3)`.
- `malloc.c:` Källkod (funktionsdefinitioner) till funktionerna enl. ovan.

I gränssnittet ingår att den returnerade minnesarean skall kunna innehålla alla typer av värden. Olika typer av variabler får bara ligga på minnesadresser som är multipler av variabelns storlek[2]. Att minnesutrymmet är organiserat på detta sätt kallas för alignment och beror på att det är mycket lättare att bygga snabba datorer när alla minnesreferenser sker så. Största datatypen i C brukar vara `double`. Storlekar anges i multipler av storleken på typen `char` som på de flesta maskiner motsvarar en byte.

När man kompilerar ditt program skall man kunna välja vilken sökordning `malloc()` (och `realloc()`) skall använda genom att på kommandoraden definiera symbolen `STRATEGY`. Tabellen nedan anger vilken strategi som skall väljas för olika värden på symbolen `STRATEGY`.

Värde på `STRATEGY`	Sökstrategi
1	First Fit
2	Best Fit
3	Worst Fit
4	Quick Fit

Man skall alltså kunna välja Quick fit med 5 snabblistor genom att kompilera med följande flaggor:

```
datorn> gcc -DSTRATEGY=4 -DNRQUICKLISTS=5 -c malloc.c
```

(Denna kommandorad kompilerar malloc.c till malloc.o och länkar alltså inte in något testprogram. Du kan naturligtvis även kombinera denna flagga med länkning. Se nedan!)

Du får gärna implementera fler strategier!

4.0 Tillägg till den obligatoriska rapporten (se Lab-PM)

Om du implementerat fler strategier än de fyra obligatoriska skall du i din rapport tala om vilka värden på `STRATEGY` man skall ange för att aktivera dina eminenta utvidgningar.

2. På en del datorarkitekturer krävs inte alignment till multipler av variabelns storlek. T.ex krävs bara alignment till 4 byte-multipler på SUN.

4.1 Tester som skall klaras av

Eftersom det är svårt att komma på alla fall under vilka ett biblioteksanrop kan göra fel, så har vi skrivit testprogram som försöker vara så elaka som möjligt, så att du lättare kan hitta fel i ditt verk. Alla testprogram är ganska pratiga, så att du kan gissa vad som pågår bakom kulisserna. Fel yttrar sig genom plötslig terminering (Segmentation Fault, Bus Error mm.) eller genom att testprogrammet skriver ut ett felmeddelande som alltid inleds med testens namn samt innehåller ordet ”`ERROR`”.

Följande testprogram finns i kurskatalogen under `labs/malloc` (se kursinformationen).

1. Filen `tstmalloc.c` är bastestet och kontrollerar om `malloc()` och `free()` verkligen allokerar minne, samt om man kan använda detta minne. Om `tstmalloc.c` rapporterar fel, är det ingen idé att försöka med de efterföljande testerna.
2. Filen `tstmerge.c` testar att du kan allokera, avallokera och slå ihop stora datablock på ett korrekt sätt (utan att det går åt minne, s.k. minnesläckor).
3. Filen `tstrealloc.c` gör motsvarande tester som `tstmalloc.c`, fast med `realloc()`. Dessutom gör den lite knepigare kombinationer av `realloc()` som har visat sig vara mindre självklara.
4. Filen `tstextreme.c` gör extremt många `malloc()` och `free()` för att upptäcka eventuellt minnesläckage, d v s att `free()` inte ger tillbaka allt minne man fick av `malloc()`. Om du får problem här skall du kontrollera att du verkligen bygger på den fria listan i `free()` samt använder den fria listan i `malloc()`.
5. Filen `tstmemory.c` undersöker minnesbehovet av din `malloc()` jämfört med hur mycket minne man begär av `malloc()`.
6. Filen `tstalgorithms.c` testar om dina algoritmer är rimliga. Den genomför ett större antal slumpvisa allokeringar och avallokeringar med `malloc()`, `realloc()` och `free()`.

Observera att de här testprogrammen i första hand testar korrektheten hos dina implementationer av minneshanteraren. Får du problem i något av testprogrammen rekommenderar vi dig att läsa koden för tetprogrammet och att köra en debugger för att hitta felen i din kod.

Eftersom testprogrammen testar korrekthet snarare än prestanda hos minneshanteraren är de kanske inte direkt lämpliga att använda i utvärderingen av din kod. Vissa kan dock med enklare modifieringar göras användbra även för det ändamålet.

4.2 Utvärdering

Du skall genomföra en *grundlig* prestandautvärdering av ditt arbete. Prestandautvärderingen skall uppfylla de grundkrav man bör ställa på vilken prestandautvärdering som helst:

- ***att man mätt rätt saker (både vad gäller att man mätt det som är relevant och att man i mätningen verkligen mätt vad man tror sig mäta)***
- ***att man vet och kan ange vilken noggrannhet man har på mätningarna*** *(just denna del ser vi lite mellan fingrarna med i det här fallet för att det inte skall bli allt för tidsödande för er)*
- ***att man valt experiment och dokumenterat dem så att de kan återupprepas (reproduceras)***

I utvärderingen skall ingå *hastighet och minnesutnyttjande* för de olika algoritmerna jämfört med systemets malloc. Du skall förklara val av utvärderingsmetod och varför (eller varför inte) dina resultat ser ut som de gör och om de är rimliga. Till din hjälp har du:

Våra testprogram. Testprogram för minnesåtgång finns i kursbiblioteket (se Sektion 4.1). Dessa testprogram testar i första hand korrektheten hos dina algoritmer. De är inte nödvändigtvis bra för prestandautvärderingar utan modifieringar eller att man noga tänker efter hur man använder dem.

Du skall själv välja lämpliga experiment för utvärderingen och testprogrammen behöver alltså inte vara de vi använder för korrekthetstesterna.

Kompileringsptimeringar. För att göra så rättvisande prestandautvärdering gentemot systemets implementation av `malloc()` mfl. bör du naturligtvis se till att kompilatorn genererat så effektiv kod som möjligt. För fallet att du använder `gcc` bör du alltså slå på full optimering genom att ange flaggan `-O4` vid kompileringen.

Profilerande bibliotek. Du måste kompilera om alla ingående filer ifall du vill profilera ett program. Använd `"gcc -O4 -pg"` för profilering. När du kör ett program som profileras, kommer du att få en fil `gmon.out` som i sin tur används av kommandot `"gprof -a`" för att få fram mycket information. Ur den informationsmängden skall du sammanställa lämpliga delar som visar hur snabba dina alster är. Använd systemets `malloc()` som exempel på en bra `malloc()` att jämföra med. Glöm inte att göra alla utskrifter villkorliga så att du kan ta bort dessa när du profilerar.

`time``(1)`. `/usr/bin/time mittprog` levererar en grov uppskattning av hur mycket tid programmet tar på sig. Observera att det finns ett kommando `time(1)` som är inbyggt i kommandotolken ifall man använder `csh(1)` eller `tcsh(1)`.

Observera att vi inte kräver att du skriver en bättre `malloc()` än den som finns i systemet. Det har lagts ner mycket möda av många personer för att få den snabb. I skrivande stund är det ganska vanligt att systemets `malloc()` underhåller flera hashlistor för olika storlekar av block som man kan tänkas allokera. Senaste skriket just nu är optimerande versioner av `malloc()` som förbättrar sig efter exekveringsprofilen av det program som använder funktionen (jfr. quick-fit).

4.2.1 Vad som skall ingå i utvärderingen

Du skall jämföra samtliga dina algoritmer mot systemets implementation vad avser exekveringstid och minnesförbrukning med avseende på:

- Bästa och värsta fall
- Rimligt/ga användningsfall

Dessutom måste du:

- Beskriva och motivera dina val av tester
- Beskriva testfallen så väl att de kan upprepas av någon utomstående
- Sammanställa och redovisa dina resultat på ett format som underlättar för läsaren att ta till sig informationen (genom t.ex grafer eller tabeller)
- Ange mätnoggranhet/fel i dina mätningar, där så är befogat, som varians eller konfidensintervall (dvs statistiskt kunna säkerställa mätresultaten).

5.0 Tips

5.1 Tänk efter före!!!!!

Innan du börjar programmera bör du tänka efter noga hur dina datastrukturer skall se ut och vilka funktioner du kan behöva för att t.ex sätta in eller ta ut block ur listor, slå samman block, dela upp block etc. Att leta efter fel i listor som man bygger på så här låg nivå är inte enkelt. Därför är det här är en uppgift där det verkligen lönar sig att arbeta strukturerat, för det är mycket enklare att verifiera mindre funktioner och sedan använda dessa än att skriva ostrukturerad kod. En stark rekommendation är att utgå från exempelkoden i “The C Programming Language” se Sektion 5.5 (*Observera att du inte får utgå från andra implementationer än just denna*). Utgår du från den

behöver du bara lägga till ett 20-tal rader kod för att få first-, best- och worst-fit att fungera.

Dettål att upprepas igen: För att kunna lösa den här uppgiften på ett bra sätt gäller att modularisera och ***återanvända*** kod. Detta för att du ska minimera kodmängden du måste skriva och för att du bara ska behöva avlusa koden en gång. Att återanvända kod är dock *inte* detsamma som att kopiera stora kodmängder och lägga in dem på nya ställen i källkodsfilen. Det leder bara till problem inte minst genom att koden blir mer svårläst, större och i värsta fallet kopierar man upp fel. Som exempel på att det kan löna sig att tänka till före kan tas att `realloc()` kan implementeras på c:a 5 rader om man gör det enkelt för sig. En sådan implementation är betydligt enklare att avlusa (och oftast effektivare/snabbare) än de exempel jag ofta har fått se som innehållit hundratals rader med kod.

Får du ändå problem eller fel så gäller det att ta till logiskt tänkande kombinerat med användande av en debugger för att komma vidare. Att debugga med utskrifter från programmet kan vara knepigt (se nedan).

5.2 Implementation

Tänk på speciella fall som kan inträffa under exekvering, som du måste hantera korrekt:

```
/* detta skall fungera korrekt */
p = malloc(0);
free(p);
free(NULL);

/* detta skall ge tillbaka p == NULL */
#include <sys/resource.h>
struct rlimit r;
getrlimit(RLIMIT_DATA, &r);
p = malloc(2 * r.rlim_max)

/* och dessa? */
p = realloc(NULL, 17);
p = realloc(p, 0);
p = realloc(NULL, 0);
```

5.3 Kompilering

Ett krav vi ställer på din kod är att kompilering skall vara möjlig att göra med den Makefile vi tillhandahåller - allt för att vi på ett enkelt och halvautomatiserat sätt skall kunna testa din kod. En sak att tänka på är att det är viktigt (även om du kompielarar på andra sätt under utvecklingen av labben), **att filen med dina biblioteksfunktioner nämns först**, så att dessa och inte systemets kommer till användning.

```
datan> gcc -o tst -g -Wall -DSTRATEGY=1 malloc.c tst.c
```

5.4 Vilka funktioner använder malloc()?

Flera av de fördefinierade C-biblioteksfunktionerna använder `malloc(3)`, detta gör att det kan vara svårt att felsöka/avlusa dina egna versioner av minneshanteringsfunktionerna. Till exempel använder `printf(3)` sig av `malloc`. Vill man ändå använda sig av felutskrifter är det trots allt ganska enkelt att via systemanrop skriva direkt till en viss fil/ström som t.ex `stderr`.

5.5 Enkelt exempel

Ett enkelt exempel på minneshanterare (ej fullständigt) som kan vara lämpligt att utgå från finns i boken och finns tillgängligt i filen `malloc.c` i katalogen för labben:

Kernighan, Richie: “The C Programming Language”, andra upplagan, Prentice Hall, sidorna 185-189. Observera att det här är den enda tillåtna exempelkoden att utgå från. Koden är skriven av experter på C och är inte helt enkel att förstå, vilket trots allt är nödvändigt att göra om man utgår från den koden.. Erfarenheten visar att man lär sig bättre hur den fungerar om man själv tvingas skriva den utgående från boken - därför finns den heller inte upplagd i någon kurskatalog.

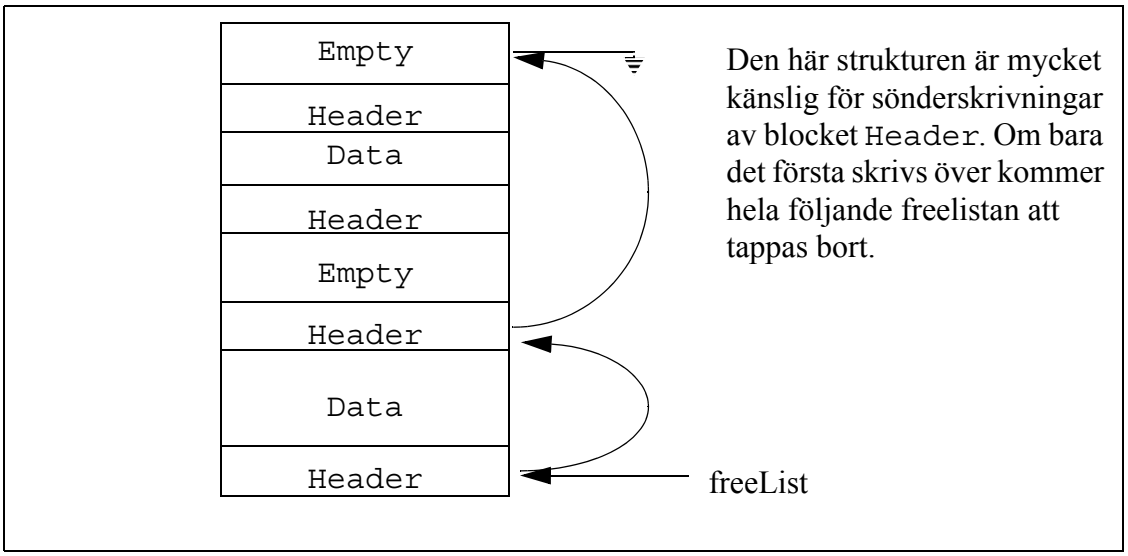
Utgår du från det exemplet så är det två saker du bör sätta dig in i och förstå grundligt:

Man använder en union som datastruktur för den header man lägger i varje block. Headern används bland annat för att länka samman lediga block i listan med fria block och för att hålla storleken på blocket (dvs storleken på den area som kan användas av användaren). Det du bör förstå är varför man använder en `union` och inte bara en ren `struct`.

Det andra som kan vara lite klurigt att förstå, men som är nödvändigt att ha koll på, är vilken enhet som används i Kernighan/Ritchies kod för att ange storleken på blocken, hur stor den är och varför man använder den och inte bara räknar storleken direkt i bytes.

6.0 Överkurs (frivilligt)

1. Er implementation bör vara robust mot sönderskrivning av kontrollstrukturer och pekare i frilistan och även mot sönderskrivning av block som t. ex. innehåller info om blockstorlek. Det bör inte heller vara möjligt att frigöra minne m.h.a. `free()` om detta minne inte kommer från `malloc()`. Notera att en lösning med kontrollstrukturer mellan block är mycket känslig för störningar (se FIGUR 2.).



FIGUR 2. Känslig minnesstruktur

2. Filerna `tstcrash_simple.c` och `tstcrash_complex.c` är elaka test som försöker att luras genom att använda `malloc()` och `free()` på fel sätt. Du hittar dem i en underkatalog till labben. Om du har gjort en `malloc()` som påstår sig vara felsäker (enligt punkt 1 ovan) behöver du klara av detta.
3. Utvidga biblioteket så att det är förberett för att integreras i en flertrådsmiljö. Koden skall vara så skriven att portering till en sådan arkitektur blir lätt, vilket innebär att du lägger till funktionera `criticalEnter()` och `criticalExit()` där dessa behövs för att skydda datastrukturer från samtidig åtkomst. `criticalEnter()` och `criticalExit()` kan däremot vara tomma makron när man inte använder biblioteket för flertrådade applikationer. Antalet kritiska sektioner och längden på dessa bör så långt som möjligt begränsas av effektivitetsskäl.
4. Implementera andra finesser. Du kan låta dig inspireras av Paul Wilsons nästan kompletta översikt av minnesallokeringsforskningen fram till 1995 (ungefär). Denna 78 sidiga skrift finns i Postscriptformat i katalogen `doc/` i kurskatalogen!

7.0 Appendix - utdrag ur ISO/IEC 9899:1990

Finns på två separata sidor.

8.0 Om rapporten

Rapporten skall vara utformad i enlighet med *"Allmänna instruktioner för LabPM"* som du hittar på kursens hemsida. Speciellt bör du tänka på att ha med:

- Ifyllt försättsblad
- En relativt utförlig beskrivning av uppgiften, hur du valt att lösa den och ditt program
- Svar på *samtliga* förberedelsefrågor från detta labPM
- Resultat från testkörningar
- Väl kommenterad källkod
- Uppgift om var källkoden finns och hur den skall kompileras (se till att vi kan accessa den!)

För att vi skall kunna förbättra labben och labbkursen vill vi också att du ägnar lite tid åt att också skriva in följande i din labbrapport:

- En uppskattning på hur mycket tid du lagt ned på att göra labben
- Betygssätt labPM, svårighetsgrad på labben och vad du fått ut av den på en skala 1-5
- Kan vi förbättra labben? Kom gärna med konstruktiv kritik!