Framsida för labrapport

Operativsystem, ID220 6

Period 2, läsår 20 13

Fyll i alla uppgifter!

Labnr.	Labnamn
3	Minneshantering

Efternamn, förnamn	Personnummer	*Tydlig* datorpostadress
Cederlund, Mattias	920926-2410	mcede@kth.se

Inlämningsdatum: 2013-12-20

Kommentarer

För internt bruk

Godkänd	Komplettera	Meddelad	Datum	Signatur

Registrerad	Ny	Gammal	G/B/U

Problembeskrivning

Uppgiften bestod i att skriva ett bibliotek med minneshanteringsfunktionerna malloc(3), free(3) och realloc(3) istället för de fördefinierade funktionerna i standardbiblioteket.

Malloc ska implementeras med flera olika metoder. First-, best-, worst- och quick-fit. I de tre förstnämnda implementationerna hålls alla lediga block i en länkad lista. First-fit allokerar första påträffade block som är nog stort. Best-fit tar reda på vilket block i listan som är närmast storleken man vill allokera som möjligt och väljer det. Worst-fit letar reda på det största blocket och allokerar det.

Quick-fit är ett försök till en optimering där man håller block av exakta storlekar, förslagsvis tvåpotenser, i en array av länkade listor. Man ska vid kompilering kunna välja hur många sådana quicklists som ska användas. Block större än det största i quicklistorna allokeras med hjälp av first-fit.

Till uppgiften hör även att göra en prestandautvärdering av de implementerade algoritmerna vad avser exekveringstider och minnesförbrukning i jämförelse med systemets algoritmer.

Programbeskrivning

Jag utgick ifrån Kernighan & Richie's kod från kurswebben när jag konstruerade min implementation av algoritmerna. För alla implementationer gäller då att man först beräknar hur många multipler av header-storleken som behöver allokeras. Det gäller dock att om storleken som anges som parameter är noll returneras null direkt.

För first-fit gäller att man loopar igenom listan med lediga block. Om ett block som är tillräckligt stort undersöker man om det är exakt nog stort. I så fall returneras det och tas bort från listan av lediga block. Om blocket är för stort uppdaterar man blockets headerinformation och på så sätt krymper blocket nedifrån. Den nedre delen av blocket, nu precis lagom stort, returneras.

Best-fit loopar igenom listan och hittar det block som är nog stort och så nära storleken av det block man vill allokera som möjligt. När det hittats utförs returnering av blocket på samma sätt som i first-fit.

För worst-fit gäller samma tillvägagångssätt som i best-fit med den enda skillnaden att man nu istället försöker hitta det största blocket i listan.

Om inget passande block hittas anropas funktionen morecore som använder sig av mmap eller sbrk för att utöka storleken av heapen och på så sätt få ett nytt minnesblock som kan allokeras. Morecore returnerar en pekare till det nya minnesblocket som man sedan anropar free på för att lägga till blocket på rätt plats i listan av lediga block. Man loopar sedan igenom listan av lediga block igen, hittar det nya blocket och allokerar det som vanligt.

Vid allokering med quick-fit undersöker vi först om blocket man vill allokera är nog litet för att finnas i någon av de quicklists som finns. Är så fallet kollar man upp vilken av quicklistorna blocket ska hämtas ur. Finns det block i korrekt quicklista så retuernas blocket. Om det inte finns något block av rätt storlek behöver man anropa morecore för att få ett nytt block från operativsystemet. Är blocket man vill få via morecore mindre än den förutbestämda minimistorleken returneras en pekare till ett minnesblock av minimistorleken. Detta block delar vi då upp i ett antal mindre block av samma storlek som vi vill allokera. Det är en av anledningarna till att blockstorlekarna i quicklistorna liksom minimistorleken fördelaktigheten är av tvåpotenser. Då går det att dela blocket jämnt. Efter uppdelning i mindre delar anropas free på blocken så att de hamnar i korrekt quicklista. Sedan tas ett block av rätt storlek ut och returneras till användaren.

Om blocket som ska allokeras är större än det största blocket i quicklistorna används first-fit, som beskrivet ovan.

För free används Kernighan & Richie's kod förutom i fallet då blocket som ska återlämnas till listan av lediga block är av en storlek som passar i quicklistorna, om metoden quick-fit används. Isåfall kollar man först om blocket är nog litet för att kunna passa i en quicklista. Är så fallet undersöks vilken lista blocket ska läggas till i varpå det läggs i korrekt quicklista.

Är blocket däremot större än maxstorleken i quicklostorna, passar inte eller om annan metod än quick-fit används är det Kernighan & Richie's kod som gäller. En mer exakt förklaring av vad den gör, rad för rad, finns i programkommentarerna.

Mer övergripande är dock att man letar reda på korrekt plats i listan av lediga block som blocket ska läggas på, listan är nämligen sorterad efter minnesadresser.

Här är även hopslagning av block implementerad. Den går till så att om det blocket som ska återlämnas passar till blocket efter i listan slås dessa två block ihop genom att man utökar det blocket som ska återlämnas storlek med det efterkommande blockets storlek samt flyttar blockets pekare till det efterkommande blockets nextpekare.

Sedan undersöks om blocket ligger direkt efter det föregående blocket. I så fall görs liknande operationer som beskrivet i fallet då nästkommande block passar direkt efter, men operationerna utförs på det föregående blocket.

Om blocken inte går att slå ihop sätts deras nextpekare på korrekt sätt. Föregående blocks nextpekare på det nya blocket och det nya blockets nextpekare på det block det föregående blockets nextpekare pekar ut.

Realloc är implementerad så att om pekaren som skickas som argument är null motsvarar det en malloc. Är storleken noll, motsvaras det av en free på utpekat block. Annars hämtas headern ut från blocket som ska reallokeras och man hämtar sedan ut storleken på det gamla blocket. Sedan allokeras ett nytt block av storleken angiven i argumentet och man kollar om det gamla blocket är mindre än det nya. Är så fallet ska memcopy anropas på det gamla blockets längd, annars på det nya. Det gamla blocket freeas till slut och det nya blocket returneras.

Förberedelsefrågor

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)

Detta varierar beroende på blockets storlek eftersom headern tar 16 bytes. Om vi räknar på det när blockstorleken går mot oändligheten och vi kan försumma headerns storlek gäller dock att i värsta fall, när 1 byte mer än någon quicklistas storlek slösas 50% bort. I medelfallet är hälften av den del som överstiger halva (nästa storlek mindre i quicklistorna) fullt. Alltså slösas då 25% bort.

Medelfallet testade jag även experimentiellt. Jag allokerade block av slumpmässig storlek som alla var nog små för att tillhöra en quicklista. Resultatet blev att 74,94 ± 0,45 % av minnet utnyttjades. (95%-igt konfidensintervall.)

Du skall i din rapport förklara all den kod i malloc.c som endast kompileras om macrot MMAP är definerat.

I morecore hämtas först heapens gräns med hjälp av sbrk(0). Sedan beräknas antalet pages som

mmap ska allokera. Man anropar mmap och får tillbaka en pekare till ett minnesutrymme som nu är tillgängligt för användning av malloc. nu sätts till antalet header-multipler som faktiskt allokerats och __endHeap-pekaren flyttas till den nya heap-gränsens adress.

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?
__endHeap skulle då bli en nullpekare.

Vad händer om man skickar NULL som första parameter till mmap(2)? Vad fungerar/fungerar inte? Vet man då var minnet reserveras?
Man vet inte var i minnet blocket kommer allokeras. Dock så är inte startvärdet som skickas in till mmap mer än en hint var blocket kommer allokeras utan den exakta adressen ges som retur.

Vad händer om man byter ut/tar bort flaggan MAP_SHARED i anropet till mmap(2)?
MAP_SHARED är en flagga som gör att mappningen kan användas av flera processer. Gissningsvis blir inte malloc allt för optimal om varje block som mmap returnerar endast kan avändas av en process.

Prestandautvärdering

För den analytiska delen av utvärderingen har jag valt att göra en kodgenomgång och försöka identifiera tidskomplexiteten och intuitivt förklara minnesförbrukningen för de olika metoderna.

Vad gäller first-fit så görs en linjär traversering av minnesblocken, men man avslutar direkt när ett lämpligt block hittats. Tidskomplexiteten kan beskrivas av O(n). I bästa fall hittar man ett block direkt, och i värsta fall hittar man inget lämpligt block och har då traverserat hela listan. Medelfallet blir då att man gått igenom halva listan innan ett block hittas.

För best- och worst-fit gäller att man alltid går igenom alla lediga block vid varje allokering. Både bästa, värsta och medelfallet här är alltså att hela listan traverseras och därmed borde dessa metoder vara mer tidskrävande än first-fit. Tidskomplexiteten är av storleksgraden O(n).

Quick-fits tidskomplexitet är mer svårbedömd då den beror mer starkt på hur stora block man vill allokera. Om blocken är stora och antalet quicklists är litet kommer first-fit att användas, och då blir tidskomplexiteten som beskrivet ovan. Däremot om blocken är nog små så kommer ett block hittas på en konstant tid, en tidskomplexitet av storleks grad O(1). Därför är det vettigt att antalet quicklists är så stort att en stor del av de block man vill allokera tas därifrån.

Dock bör man nämna att alla metoder kommer att behöva använda mmap/sbrk för att utöka storleken på heapen någon gång. Denna operation har jag inte tagit någon hänsyn till i tidigare beräkningar av tidskomplexiteten.

Vad gäller minnesförbrukning så kommer first-fit, best-fit och worst-fit att alltid allokera exakt lagom stora block. Det enda minnesslösande som sker här är att det är möjligt att få kvar block i listan av lediga block som är så små att de inte är användbara, extern fragmentering. Detta kommer ske i alla tre metoder men minst av best-fit eftersom man försöker se till att dessa block blir så små som möjligt. Worst-fit blir av den anledningen sämst av de tre, och first-fit borde hamna någonstans i mellan.

Quick-fit har som redan nämnt ett minnesutnyttjande på 25% i medelfallet då block tas från quicklistorna och bör således vara sämre än övriga metoder när det gäller minnesförbrukning. Här har vi istället en intern fragmentering. Om antalet block som är större än det största i quicklistorna

ökar, så ökar också minnesutnyttjandet, men vi förlorar istället i tid. Här bör en avvägning göras för en bra balans mellan exekveringstid och minnesutnyttjande.

För att utföra en experimentell utvärdering av prestandan av algoritmerna valde jag att skriva om det givna tstalgorithms.c. I det utförande som användes för testningarna allokeras först 10000 block av slumpvis storlek, men med maxstorleken 1024 * sizeof(double). Därefter reallokeras 10000 slumpvis utvalda block till en storlek av mellan 0 och 2048 * sizeof(double). Till sist anropas free på alla block. Mellan varje del i exekveringen av testprogrammet görs tidsmätningar och minnesmätningar som redovisas senare i rapporten. Tidsmätningarna är gjorda med systemanropet gettimeofday, vilket ger en noggrannhet ner på mikrosekunder.

Att jag valde att använda slumpvis storlek på de allokerade blocken är eftersom jag tänker mig att en användarprocess allokerar precis så stora block den behöver, vilket kan variera på grund av indata eller liknande. Det kan även tänkas att flera processer kör samtidigt, och de i sin tur inte alla allokerar block av samma storlek.

Testerna är utförda på en maskin utan några övriga körande processer och alla tester är körda med samma förutsättningar. Specifikationerna på hårdvaran är inte väsentlig eftersom testerna endast är till för att jämföras med varandra och inte med någon annans testkörningar. Nedan följer data från testkörningar. Testprogrammet i sig är kört 10 gånger efter varandra för varje allokeringsmetod och utifrån det har 95%-iga konfidensintervall beräknats.

Testerna är kompilerade med följande kommando:
gcc -O4 -DSTRATEGY=x -DNRQUICKLISTS=9 -o test malloc.c test.c
där x är vilken strategi man vill använda sig av, definierat som i labPM.

För systemets inbyggda malloc gäller:
gcc -O4 -o test_sys test_sys.c

För quick-fit gäller att testet är utfört med NRQUICKLISTS=9. På så sätt kommer hälften av blocken allokeras från quicklistorna. Med ett mindre antal quicklistor motsvaras metoden av first-fit, vilket inte är intressant för utvärderingen.

Tabell 1, testkörningar av metoderna

	First	Best	Worst	Quick	Sys
Tidsåtgång malloc (s)	2,107 ± 0,032	2,118 ± 0,046	3,993 ± 0,060	1,408 ± 0,011	0,034 ± 0,001
Minnesutnyttjande malloc (%)	95,73 ± 0,80	99,14 ± 0,67	90,55 ± 0,53	84,42 ± 0,47	99,59 ± 0,29
Tidsåtgång realloc (s)	4,126 ± 0,038	4,363 ± 0,052	7,782 ± 0,069	2,786 ± 0,017	0,049 ± 0,002
Minnesutnyttjande realloc (%)	84,44 ± 0,60	89,42 ± 0,67	72,43 ± 0,81	79,79 ± 0,56	86,74 ± 0,22
Tidsåtgång free (s)	1,194 ± 0,013	1,066 ± 0,014	1,370 ± 0,017	1,088 ± 0,017	0,011 ± 0,000

Eftersom tidsmätningarna utförts på alla operationer (av samma typ) och inte på enskilda så testade jag även tidsåtgången för testprogrammets kontrollstrukturer. Tidsåtgången blev då 0,00057 ± 0,00020 sekunder, vilket jag anser vara försumbart. Även detta test utfördes 10 gånger och resultatet är angivet som ett 95%-igt konfidensintervall.

För att sedan kunna utvärdera mätfelens storlek har även alla tester körts med samma startfrö till slumptalsgeneratorn. För alla testkörningar användes nu fröet 1234567890. Det som kan konstateras av detta test är att detta gav värden från slumpgeneratorn som var svårare för alla metoder samt att det 95%-iga konfidensintervallets storlek var större för alla metoder (utom first och worst vilka ändå var inom samma storleksgrad) vilket skulle tyda på att det snarare är mätfelen är den dominerande felkällan, än känsligheten

för indata.

Tabell 2, testkörningar med samma frö till slumptalsgeneratorn

	First	Best	Worst	Quick	Sys
Storlek av konfidenintervall för tidsåtgång malloc (s)	± 0,024	± 0,080	± 0,025	± 0,031	± 0,002

Utifrån de experimentella resultaten kan vi se att de stämmer bra överens med de modeller som sattes upp under den analytiska delen av utvärderingen. Inga experimentella resultat avvek i större utsträckning från de analytiska, därmed anser jag dem verifierade.

All rådata från testerna finns att tillgå i exceldokumentet på följande plats: /afs/ict.kth.se/home/m/c/mcede/os/lab3 där filen heter testresults.xls.

Resultat från testkörningar

First-fit:

```
[mcede@avril lab3]$ ./t0
-- Testing merging of deallocated large blocks ( >= 16384 bytes)
Test passed OK

[mcede@avril lab3]$ ./t1
./t1, line 69: -- This test checks malloc(), free() and realloc()
./t1: Max memory allocated 11889352
./t1: Memory consumed 14450688

[mcede@avril lab3]$ ./t2
./t2, line 23: -- This test will search for memory leaks
./t2, line 24: At most 3.0x pages are allocated and recycled
./t2: Used memory in test: 0x4000 (= 4.00 * pagesize)

[mcede@avril lab3]$ ./t3
./t3, line 19: -- Test malloc() for unusual situations
./t3, line 21: Allocate small block of 17 bytes
./t3, line 25: Write on allocated block
./t3, line 28: Allocate big block of 4711 bytes
./t3, line 32: Write on allocated block
./t3, line 34: Free big block
./t3, line 37: Free small block
./t3, line 40: Free NULL
./t3, line 43: Allocate zero
./t3, line 47: Free pointer from malloc(0)
./t3, line 50: Test alignment for double

[mcede@avril lab3]$ ./t4
./t4, line 32: Testing memory utility
./t4, line 40: Getting small pieces of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using 2.00 times worst case calculation
```

```
./t4, line 66: Small memory handled OK
./t4, line 69: Getting big blocks of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4, line 88: Allocations versus worst case memory usage:
./t4: Using 1.12 times worst case calculation
./t4, line 96: Big memory handled OK

[mcede@avril lab3]$ ./t5
./t5, line 17: -- Test realloc() for unusual situations
./t5, line 19: Allocate 17 bytes with realloc(NULL, 17)
./t5, line 24: Write on allocated block
./t5, line 26: Increase block size with realloc(., 4711)
./t5, line 33: Write on allocated block
./t5, line 35: Decrease block size with realloc(., 17)
./t5, line 40: Decreased block size
./t5, line 43: Free block with realloc(., 0)
./t5, line 48: realloc(p, 0) returns null pointer
./t5, line 49: Free pointer allocated with realloc(NULL, 0)
```

Best-fit:

```
[mcede@avril lab3]$ ./t0
-- Testing merging of deallocated large blocks ( >= 16384 bytes)
Test passed OK

[mcede@avril lab3]$ ./t1
./t1, line 69: -- This test checks malloc(), free() and realloc()
./t1: Max memory allocated 11781912
./t1: Memory consumed 12541952

[mcede@avril lab3]$ ./t2
./t2, line 23: -- This test will search for memory leaks
./t2, line 24: At most 3.0x pages are allocated and recycled
./t2: Used memory in test: 0x4000 (= 4.00 * pagesize)

[mcede@avril lab3]$ ./t3
./t3, line 19: -- Test malloc() for unusual situations
./t3, line 21: Allocate small block of 17 bytes
./t3, line 25: Write on allocated block
./t3, line 28: Allocate big block of 4711 bytes
./t3, line 32: Write on allocated block
./t3, line 34: Free big block
./t3, line 37: Free small block
./t3, line 40: Free NULL
./t3, line 43: Allocate zero
./t3, line 47: Free pointer from malloc(0)
./t3, line 50: Test alignment for double

[mcede@avril lab3]$ ./t4
./t4, line 32: Testing memory utility
./t4, line 40: Getting small pieces of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
```

```
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using 2.00 times worst case calculation
./t4, line 66: Small memory handled OK
./t4, line 69: Getting big blocks of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4, line 88: Allocations versus worst case memory usage:
./t4: Using 1.12 times worst case calculation
./t4, line 96: Big memory handled OK

[mcede@avril lab3]$ ./t5
./t5, line 17: -- Test realloc() for unusual situations
./t5, line 19: Allocate 17 bytes with realloc(NULL, 17)
./t5, line 24: Write on allocated block
./t5, line 26: Increase block size with realloc(., 4711)
./t5, line 33: Write on allocated block
./t5, line 35: Decrease block size with realloc(., 17)
./t5, line 40: Decreased block size
./t5, line 43: Free block with realloc(., 0)
./t5, line 48: realloc(p, 0) returns null pointer
./t5, line 49: Free pointer allocated with realloc(NULL, 0)
```

Worst-fit:

```
[mcede@avril lab3]$ ./t0
-- Testing merging of deallocated large blocks ( >= 16384 bytes)
Test passed OK

[mcede@avril lab3]$ ./t1
./t1, line 69: -- This test checks malloc(), free() and realloc()
./t1: Max memory allocated 12121656
./t1: Memory consumed 18259968

[mcede@avril lab3]$ ./t2
./t2, line 23: -- This test will search for memory leaks
./t2, line 24: At most 3.0x pages are allocated and recycled
./t2: Used memory in test: 0x4000 (= 4.00 * pagesize)

[mcede@avril lab3]$ ./t3
./t3, line 19: -- Test malloc() for unusual situations
./t3, line 21: Allocate small block of 17 bytes
./t3, line 25: Write on allocated block
./t3, line 28: Allocate big block of 4711 bytes
./t3, line 32: Write on allocated block
./t3, line 34: Free big block
./t3, line 37: Free small block
./t3, line 40: Free NULL
./t3, line 43: Allocate zero
./t3, line 47: Free pointer from malloc(0)
```

```
./t3, line 50: Test alignment for double

[mcede@avril lab3]$ ./t4
./t4, line 32: Testing memory utility
./t4, line 40: Getting small pieces of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using 2.00 times worst case calculation
./t4, line 66: Small memory handled OK
./t4, line 69: Getting big blocks of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4: Using total of 0x45b000 of memory
./t4, line 88: Allocations versus worst case memory usage:
./t4: Using 1.12 times worst case calculation
./t4, line 96: Big memory handled OK

[mcede@avril lab3]$ ./t5
./t5, line 17: -- Test realloc() for unusual situations
./t5, line 19: Allocate 17 bytes with realloc(NULL, 17)
./t5, line 24: Write on allocated block
./t5, line 26: Increase block size with realloc(., 4711)
./t5, line 33: Write on allocated block
./t5, line 35: Decrease block size with realloc(., 17)
./t5, line 40: Decreased block size
./t5, line 43: Free block with realloc(., 0)
./t5, line 48: realloc(p, 0) returns null pointer
./t5, line 49: Free pointer allocated with realloc(NULL, 0)
```

Quick-fit (NRQUICKLISTS=5):

```
[mcede@avril lab3]$ ./t0
-- Testing merging of deallocated large blocks ( >= 16384 bytes)
Test passed OK

[mcede@avril lab3]$ ./t1
./t1, line 69: -- This test checks malloc(), free() and realloc()
./t1: Max memory allocated 11741616
./t1: Memory consumed 14376960

[mcede@avril lab3]$ ./t2
./t2, line 23: -- This test will search for memory leaks
./t2, line 24: At most 3.0x pages are allocated and recycled
./t2: Used memory in test: 0x6000 (= 6.00 * pagesize)

[mcede@avril lab3]$ ./t3
./t3, line 19: -- Test malloc() for unusual situations
./t3, line 21: Allocate small block of 17 bytes
```

```
./t3, line 25: Write on allocated block
./t3, line 28: Allocate big block of 4711 bytes
./t3, line 32: Write on allocated block
./t3, line 34: Free big block
./t3, line 37: Free small block
./t3, line 40: Free NULL
./t3, line 43: Allocate zero
./t3, line 47: Free pointer from malloc(0)
./t3, line 50: Test alignment for double

[mcede@avril lab3]$ ./t4
./t4, line 32: Testing memory utility
./t4, line 40: Getting small pieces of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using total of 0x4000 of memory
./t4: Using 2.00 times worst case calculation
./t4, line 66: Small memory handled OK
./t4, line 69: Getting big blocks of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4: Using total of 0x450000 of memory
./t4, line 88: Allocations versus worst case memory usage:
./t4: Using 1.10 times worst case calculation
./t4, line 96: Big memory handled OK

[mcede@avril lab3]$ ./t5
./t5, line 17: -- Test realloc() for unusual situations
./t5, line 19: Allocate 17 bytes with realloc(NULL, 17)
./t5, line 24: Write on allocated block
./t5, line 26: Increase block size with realloc(., 4711)
./t5, line 33: Write on allocated block
./t5, line 35: Decrease block size with realloc(., 17)
./t5, line 40: Decreased block size
./t5, line 43: Free block with realloc(., 0)
./t5, line 48: realloc(p, 0) returns null pointer
./t5, line 49: Free pointer allocated with realloc(NULL, 0)
```

Väl kommenterad kod

malloc.c

```
/*
 * Description - This is a malloc implementation that contains the three
 *                        memory allocation functions malloc, realloc and free.
 *                        Usage is the same as for the functions in the standard
library.
 *
 */

#include "brk.h"
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <stdlib.h>

#define STRATEGY_FIRST 1
#define STRATEGY_BEST 2
#define STRATEGY_WORST 3
#define STRATEGY_QUICK 4

#define SMALLEST 2                                              /*Smallest size of
quicklist blocks*/

#ifndef MAP_ANONYMOUS                                           /*Some error wtih
MAP_ANONYMOUS if not defined...*/
#define MAP_ANONYMOUS 32
#endif

#define NALLOC 512                                              /* minimum #units to
request */

#ifndef NRQUICKLISTS                                            /*If number of
quicklist is undefined, use 5*/
#define NRQUICKLISTS 5
#endif

#ifndef STRATEGY                                                /*If strategy is
undefined, use quick*/
#define STRATEGY 4
#endif

typedef double Align;                                   /* for alignment to long boundary */

union header {                                          /* block header */
      struct {
            union header *next;                             /* next block if on free list */
            unsigned size;                                  /* size of this block  - what
unit? */
      } s;
      Align x;                                          /* force alignment of blocks */
};

typedef union header Header;

static Header base;                                     /* empty list to get started */
static Header *freep = NULL;                            /* start of free list */

static Header* quicklist[NRQUICKLISTS] = {0};/*List for our quicklist memory blocks*/
```

```
/* free
 *
 * free - put block ap in the free list
 *
 * void * ap - the block to free
 *
 */

void free(void * ap)
{
	Header *bp, *p;

	if(ap == NULL) return;					/* Nothing to do
*/

	bp = (Header *) ap - 1;					/* point to
block header */

	if (STRATEGY == STRATEGY_QUICK) {
	/*Quick*/
		if (bp->s.size <= (SMALLEST << (NRQUICKLISTS - 1))) {	/*If block is
within the range of blocks we have in quicklist*/
			int i;
			for (i = 0; bp->s.size > (SMALLEST << i); i++) {	/*Loop to find
out which list to put the memory block in*/
			}
			bp->s.next = quicklist[i];
	/*Add the block to appropriate list*/
			quicklist[i] = bp;
			return;
		}
	}

	for(p = freep; !(bp > p && bp < p->s.next); p = p->s.next)/*Loop to find where in
the list of free blocks to put block*/
		if(p >= p->s.next && (bp > p || bp < p->s.next))	/*If not looped
whole list and block to free adress is larger than p*/
			break;					/* freed block
at atrt or end of arena */

	if(bp + bp->s.size == p->s.next) {			/* join to upper
nb, if block to free ends at p's next block */
		bp->s.size += p->s.next->s.size;			/*Set pb's
size to pb.size + p.size*/
		bp->s.next = p->s.next->s.next;
	/*set bp's nextpointer to p's next,nextpointer*/
	}
	else
		bp->s.next = p->s.next;
	/*Else, set bp's nextpointer to p's nextpointer*/
	if(p + p->s.size == bp) {				/* join to lower
nbr, if a block ends right before bp */
		p->s.size += bp->s.size;
	/*Add pb's size to p's size*/
		p->s.next = bp->s.next;
	/*Set p's nextpointer to bp's nextpointer*/
	} else
		p->s.next = bp;
		/*Else, set p's nextpointer to bp*/
	freep = p;
```

```
            /*Set freepointer to p*/
}


#ifdef MMAP

static void * __endHeap = 0;

void * endHeap(void)
{
       if(__endHeap == 0) __endHeap = sbrk(0);
       return __endHeap;
}
#endif

/* morecore
 *
 * morecore - ask system for more memory
 *
 * unsigned nu - size of block, in header-multiples
 */

static Header *morecore(unsigned nu)
{
       void *cp;
       Header *up;
#ifdef MMAP
       unsigned noPages;
       if(__endHeap == 0) __endHeap = sbrk(0);
#endif

       if(nu < NALLOC)
              nu = NALLOC;
#ifdef MMAP
       noPages = ((nu*sizeof(Header))-1)/getpagesize() + 1;
       cp = mmap(__endHeap, noPages*getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
       nu = (noPages*getpagesize())/sizeof(Header);
       __endHeap += noPages*getpagesize();
#else
       cp = sbrk(nu*sizeof(Header));
#endif
       if(cp == (void *) -1){                                  /* no space at all */
              perror("failed to get more memory");
              return NULL;
       }
       up = (Header *) cp;
       up->s.size = nu;
       /*free((void *)(up+1));*/ /*Not compatible with quick-malloc*/
       return up;
}

/* malloc
 *
 * malloc - Allocate a block of memory
 *
 * size_t nbytes - number of bytes of block to allocate
 *
 */

void * malloc(size_t nbytes) {
       Header *currentp, *prevp;
```

```
	Header * morecore(unsigned);
	unsigned nunits;

	Header *best = NULL, *prevbest = NULL;
	size_t value = (size_t) 0;

	if(nbytes == 0) {
	/*No memory needs to be allocated, return null*/
		return NULL;
	}

	nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) +1;	/*Number of header-
sizes we need for the block*/

	if (STRATEGY == STRATEGY_QUICK) {					/*Quick*/
		int i;
		if (nunits <= (SMALLEST << (NRQUICKLISTS - 1))) {	/*If the requested
block is smaller than the maxsize of the list*/
			for (i = 0; nunits > (SMALLEST << i); i++) { /*Loop to find out
which list to fetch memory block from*/
			}
			if (quicklist[i] != NULL) {
	/*If there already exists a block, return it*/
				Header* header = quicklist[i];
				quicklist[i] = header->s.next;
				return (void *) (header + 1);
			}
			else {
			/*Else, get blocks of correct size*/
				size_t size = (SMALLEST << i);
	/*Size of each block to allocate*/
				size_t tomorecore = size;
	/*Size of block to request from OS*/

				if (tomorecore < NALLOC) {
		/*If the block to request is smaller than NALLOC, increase*/
					tomorecore = NALLOC;
				}
				Header* newblock = morecore(tomorecore);
	/*Call morecore to get a new block form the OS*/

				if (newblock == NULL) {
		/*If no memory left, return null*/
					return NULL;
				}

				int k;
				for (k = 0; k < tomorecore; k += size) {
	/*Split the block into pieces of correct size*/
					newblock[k].s.size = size;
		/*Set the correct size of the block*/
					free((void *) &(newblock[k+1]));
	/*Free the block, putting it in the correct list*/
				}
				Header* toreturn = quicklist[i];
	/*When done splitting, get the block to return*/
				quicklist[i] = toreturn->s.next;
	/*Move the pointer to point at the next block*/
				return (void *) (toreturn + 1);
	/*Return the data-area of the block*/
			}
		}
```

```
	}
	if ((prevp = freep) == NULL) {
		base.s.next = freep = prevp = &base;
		base.s.size = 0;
	}

	for(currentp = prevp->s.next;  ; prevp = currentp, currentp = currentp->s.next) {
	/*Loop through the list of available blocks*/
		if(currentp->s.size >= nunits) {
	/*Found block is big enough */
			if (STRATEGY == STRATEGY_FIRST || STRATEGY == STRATEGY_QUICK) {
	/*First or big block in quick*/
				if (currentp->s.size == nunits) {
	/*Block is exactly the right size*/
					prevp->s.next = currentp->s.next;
				}
				else {
	/*Block is bigger, allocate tail end */
					currentp->s.size -= nunits;
					currentp += currentp->s.size;
					currentp->s.size = nunits;
				}
				freep = prevp;
				return (void *)(currentp+1);
			}
			if (STRATEGY == STRATEGY_BEST) {
				if (best == NULL || currentp->s.size <= value) {
/*Block is smaller than the old best*/
					best = currentp;
					prevbest = prevp;
					value = currentp->s.size;
				}
			}
			if (STRATEGY == STRATEGY_WORST) {
				if (best == NULL || currentp->s.size > value) {
	/*Block is bigger than the old best*/
					best = currentp;
					prevbest = prevp;
					value = currentp->s.size;
				}
			}
		}
		if(currentp == freep) {
		/* wrapped around free list */
			if ((STRATEGY == STRATEGY_BEST) || (STRATEGY == STRATEGY_WORST)) {
				if (best != NULL) {
					/*If a best match is found*/
					currentp = best;
					prevp = prevbest;
					if (currentp->s.size == nunits) {
				/*If exactly the right size*/
						prevp->s.next = currentp->s.next;
					}
					else {
		/*Block is bigger, allocate tail end */
						currentp->s.size -= nunits;
					/*Resize head block*/
						currentp += currentp->s.size;
				/*Move the pointer to the new start*/
						currentp->s.size = nunits;
					/*Set the size of the new block*/
					}
```

```
                                freep = prevp;
                                return (void *)(currentp + 1);
                        }
                }
                if((currentp = morecore(nunits)) == NULL) {
        /*No block fits, call the OS for more*/
                        return NULL;                                    /*No
available memory left left */
                }
                free((void *) (currentp+1));
        /*Free the new memory block, so it will become available*/
              }
        }
}

/* realloc
 *
 * realloc - Reallocate a block of memory
 *
 * void * ptr - Pointer of block to reallocate
 * size_t size - Size of the new block
 *
 */

void *realloc(void *ptr, size_t size) {
        if (ptr == NULL) {                                              /*if ptr =
null, its a malloc*/
              return malloc(size);
        }
        if (size == 0) {                                                /*If new
size = 0, its a free*/
              free(ptr);
              return NULL;
        }
        Header* h = ((Header*) ptr) - 1;                        /*Get the header from
the pointer*/
        size_t oldsize = (h->s.size - 1) * sizeof(Header);  /*Get the old size from the
header*/
        void * newptr = malloc(size);                                   /*Allocate a
memory block of the new size*/
        if (size < oldsize) {                                           /*If the
new size is smaller than the old one, we will only copy*/
              oldsize = size;
        /*untill the new size*/
        }
        memcpy(newptr, ptr, oldsize);                                   /*Do the copy*/
        free(ptr);
        /*Free the old block*/
        return newptr;
}
```

test.c

```
/*
 * Description - This is a program to test the functions malloc,
 *                       realloc and free for time and memory consumption.
 *
 * Usage - test [seed]
 *           If seed is specified it will use this as start-seed for
 *           random number generator, otherwise the seed is the current time.
 *
 * Save the address of the twilight zone.
 * First phase - allocate a MAXPOSTS number of memory blocks each
```

```
 *               of random size but of MAXSIZE/2.
 *
 * Second phase - Realloc MAXITERS blocks of random size between
 *                        0 and MAXSIZE.
 *
 * Third phase - Free all blocks
 *
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include "malloc.h"
#include "tst.h"
#include "brk.h"
#include <time.h>
#include <sys/time.h>
#include <sys/types.h> /*definierar typen pid_t*/

#define MAXPOSTS 10000
#define MAXSIZE  2048
#define MAXITERS 10000
#define ALLOCATED(i) (memPosts[i].size > 0)

typedef struct {
       double *ptr;
       int size;
} allocpost;

allocpost memPosts[MAXPOSTS];

void calcMemUsage(int *max) {
       int sum=0,i;
       for(i=0;i<MAXPOSTS;i++) {
              sum += memPosts[i].size*sizeof(double);
       }
       if(sum > *max) {
              *max = sum;
       }
}

int main(int argc, char *argv[]) {
       int i, maxMem=0;
       void *start, *end;
       struct timeval starttv;
       struct timeval endtv;

       if (argc > 1) {
              printf("Using seed: %s", argv[1]);
              srand(atoi(argv[1]));
       }
       else {
              srand((unsigned int)time(NULL));
       }

#ifdef MMAP
       start = endHeap();
#else
       start = (void *)sbrk(0);
#endif

       gettimeofday(&starttv, NULL);
```

```
	/*Alloc MAXPOSTS blocks of memory*/
	for(i=0;i<MAXPOSTS;i++) {
		memPosts[i].size = rand()%(MAXSIZE/2);
#ifndef NO
		memPosts[i].ptr = (double*) malloc(memPosts[i].size*sizeof(double));
#endif
	}

	gettimeofday(&endtv, NULL);

#ifdef MMAP
	end = endHeap();
#else
	end = (void *) sbrk(0);
#endif

	printf("Time for malloc: %f\n", ((float)(endtv.tv_sec-starttv.tv_sec) + (float)
(endtv.tv_usec-starttv.tv_usec)/1000000));
	calcMemUsage(&maxMem);
	printf("Maxmem after malloc: %d\n", maxMem);
	printf("Memory consumed after malloc %ld\n\n", (unsigned long)(end-start));

	gettimeofday(&starttv, NULL);

	/*Realloc MAXITERS blocks of memory*/
	for(i=0;i<MAXITERS;i++) {
		int index;
		index = rand()%MAXPOSTS;

		if(ALLOCATED(index)) {
			if(rand()%5 < 3) {
				memPosts[index].size = rand()%MAXSIZE;
#ifndef NO
				memPosts[index].ptr =  (double*) realloc(memPosts[index].ptr,
memPosts[index].size*sizeof(double));
#endif
			}
		}
	}

	gettimeofday(&endtv, NULL);

#ifdef MMAP
	end = endHeap();
#else
	end = (void *) sbrk(0);
#endif

	printf("Time for realloc: %f\n", ((float)(endtv.tv_sec-starttv.tv_sec) + (float)
(endtv.tv_usec-starttv.tv_usec)/1000000));
	calcMemUsage(&maxMem);
	printf("Maxmem after realloc: %d\n", maxMem);
	printf("Memory consumed after realloc %ld\n\n", (unsigned long)(end-start));

	gettimeofday(&starttv, NULL);

	/*Free all blocks*/
	for(i=0;i<MAXPOSTS;i++) {
#ifndef NO
		memPosts[i].size = 0;
		free(memPosts[i].ptr);
#endif
```

```
	}

	gettimeofday(&endtv, NULL);
	printf("Time for free: %f\n", ((float)(endtv.tv_sec-starttv.tv_sec) + (float)
(endtv.tv_usec-starttv.tv_usec)/1000000));

	return 0;
}
```

Var källkoden finns

Källkoden finns på /afs/ict.kth.se/home/m/c/mcede/os/lab3 där programmet heter malloc.c. Även testkoden finns på samma plats och hur den komplileras är angivet under prestandautvärdering.

Koden kompileras med given makefil där -DSTRATEGY=x och -DNRQUICKLISTS=y definieras beroende på vilka inställningar man vill kompilera med enligt labPM.

Verksamhetsberättelse och utvärdering av laborationsuppgiften

Jag utgick ifrån Kernighan & Richie's kod från kurswebben när jag konstruerade min implementation av algoritmerna. Sedan började jag implementera first-, best- och worst-fit i den ordningen och slutligen quick-fit.

Uppskattad tid lagd på laborationen: 25 timmar, varav 10 på programmet, 15 på utvärdering och rapport.
Betyg för labPM: 3
Svårighetsgrad för laborationen: 5
Vad som fåtts ut av laborationen: 3
Vad som kan förbättras/kommentarer: Eftersom labben är så löst formulerad är den mer en labb inom ingenjörsmässigt arbete än inom OS.